FORMAL VERIFICATION SUPPORT FOR ELLA

IED 4/1/1357

Deliverable D2.3b

A semantics for Core ELLA*

Version 1.5.4

DRAFT for FORMAL REVIEW

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

May 11, 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.

Contents

1 Introduction

In previous work, we have developed and explored IO automata as semantic models for a much simplified form of ELLA (called Boolean Kernel ELLA)[MDED91]. On the basis of this work, a compiler was developed that produced literal representations of these models, which were then used for verification. The experience gained in this work has given us some confidence in the underlying semantic model and encouraged us to extend it to cover the additional datatypes and language constructs required to cover first of all Kernel ELLA and now the larger subset, Core ELLA.

The semantics presented here employs the same basic automaton constructions as were used in the BKE semantics, although their presentation is somewhat different. The major difference in the models presented here is the much more complex structure of the input/output signatures. These are now 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 have also needed to present the static semantics of the language explicitly, which was not done for BKE. However, very similar work on static semantics of Core and Kernel ELLA had already been done by Mike Hill and is contained in [HM92]. This work necessarily involved presenting an effective transformation from Core to Kernel ELLA, which resulted in more complexity than would have been necessary if only the static semantics of Core ELLA had been given.

An earlier version of this document (defining semantics for Kernel ELLA[BGMW92]) has been read by John Morison and Mike Hill of DRA (RSRE), who have provided valuable comments. It should be emphasised, however, that we believe that further discussion is essential before we can have confidence that the semantics presented represents the 'understood' semantics of ELLA. It has not been our intention to change Core ELLA, but to capture its essence. We do not, however, regard minor differences, such as those at the lexical level, as significant.

Finally, it should be admitted that the presentation of the semantics in its current state is, although quite long, still somewhat terse and lacking in explanation in places. This will, of course, be remedied in future versions of this document.

Section 2 describes the approach we have adopted to presenting the semantic definitions. Section 3 discusses the general relationship between the semantics presented here and results obtained by running a simulation. Section 4 defines the underlying semantic structures needed to present the semantic definition of Core ELLA, which is given in section 5. Section 6 gives a summary and indicates future directions.

2 Overview of semantic approach

The approach adopted here extends our earlier work using IOA semantics by adding more structured forms of *data* into the semantic models. It turns out that every well-specified system definable within ELLA will be deterministic. As a result, the behavioural transformations corresponding to any system are therefore functional, where these transforms are determined by the present state and the system input. The result of each transition may then cause a change of state and possibly some further output.

A more algebraic flavour could be given to this semantics by defining primitive ELLA constructions, such as the various forms of delay operator given within ELLA, by algebraic equations over their behaviour, when considered as input/output languages.

2.1 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 layed 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 homogenous whole, by firstly defining the abstract syntax as a data type, then presenting the static semantics predicate by induction over this syntatic 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 homogenous 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 peice 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 in both ways; firstly, in a horizontal format and then in a vertically organised way within the appendices. Appendix B contains some further information about how to read the semantic definition and the notational conventions used.

3 Relationships between Semantics and Simulation

```
The following section is an draft.  It is included for the purposes of further
review by John Morison and Mike Hill, DRA Malvern.
```

The semantics presented within this document defines an *exact* mathematical semantics of Core ELLA. It is exact in the sense that it provides a precise criterion for when two ELLA function objects are behaviourally equivalent.

The main reason that simulation methods are considered is to provide timely feedback about the behaviour of a design **without** expensive semantic analysis being required. Accordingly, a simulation of ELLA definitions provides an animation of the system specified, so allowing the designer to explore the behaviours exhibited by systems for particular sequences of inputs.

Now, there are occasions where such a system may be either *underspecified*, where its behaviour is not sufficiently constrained or where the system suffers some form of erroneous behaviour. Under either of these circumstances, the simulated outputs from the system may not be completely determined and as a result, a reasonable simulator *should* then output a '?' (or similar).

This line of reasoning leads towards the following notions. So, purely for the purposes of the definitions contained in this section, we may assume that any such simulator could be thought of, in principle, as a function mapping ELLA text and input sequences to (possibly sets of) output sequences. This might be more formally stated by defining the type of all possible simulators to be:

$$
\begin{array}{l}
\mathit{sim} \in \mathit{Simulator} = (\text{ ELLA text } \times \text{ sequence } \rightarrow \mathbb{P}(\text{ sequence })) \\
\textsf{where} \\
\quad \forall\, \mathit{sys},\ \mathit{inp\text{-}seq},\ \mathit{oup\text{-}seq} \bullet \\
\qquad \mathit{oup\text{-}seq} \in \mathit{sim}(\mathit{sys}, \mathit{inp\text{-}seq}) \;\Rightarrow\; \textsf{len}\, \mathit{oup\text{-}seq} = \textsf{len}\, \mathit{inp\text{-}seq}
\end{array}
$$

assuming that the input and output sequences consist of a finite sequence of signal vectors, possibly involving occurrences of '?'. The property requires any simulator to produce outputs of equal length to the input.

In a similar way, we assume the exact semantics for Core ELLA is conveniently represented in terms of its action on *exact* signal traces (i.e. *query free* sequences of signal vectors) and which then has type:

$$
\begin{array}{l}
\mathit{semantics} : \text{ ELLA text } \times \text{ trace } \rightarrow \mathbb{P}(\text{ trace })) \\
\textsf{where} \\
\quad \forall\, \mathit{sys},\ \mathit{inp\text{-}trc},\ \mathit{oup\text{-}trc} \bullet \\
\qquad \mathit{oup\text{-}trc} \in \mathit{semantics}(\mathit{sys}, \mathit{inp\text{-}trc}) \;\Rightarrow\; \textsf{len}\, \mathit{oup\text{-}trc} = \textsf{len}\, \mathit{inp\text{-}trc}
\end{array}
$$

where the length of any output trace is equal to the length of the input trace.

Clearly, we need to discuss what the correspondance should be between a simulator and the exact semantics for Core ELLA. To do this, we first note that every sequence of signal vectors can be translated into a corresponding *set* of *query-free* traces of signal vectors. For our purposes, we informally introduce the following relation and conversion function:

$$\_\ \textbf{is covered by}\ \_\ \subseteq\ \text{trace} \times \text{sequence}$$
$$\textit{expand-sequence} : \text{sequence} \rightarrow \mathbb{P}(\text{trace})$$
$$\textit{expand-sequence}(\textit{seq}) =$$
$$\quad \{ \textit{tr} \in \text{trace} \mid \textit{tr}\ \textbf{is covered by}\ \textit{seq} \}$$

The **is covered by** relation states how query-free traces are covered by sequences that possibly contain '?', by permitting an arbitrary value (of appropriate type) to replace occurrences of ?. The function *expand-sequence* simply takes a sequence and collects together all query free traces covered by that sequence.

We now say what it means for a simulator, $\textit{sim} \in \textit{Simulator}$ to *conserves* the semantics of Core ELLA:

$$\textit{sim}\ \textbf{is semantically conservative} \quad \Leftrightarrow$$
$$\quad \forall\, \textit{sys},\ \textit{inp-seq} \bullet \forall\, \textit{inp-trc} \in \textit{expand-sequence}(\textit{inp-seq}) \bullet$$
$$\quad\quad \forall\, \textit{oup-trc} \in \textit{semantics}(\textit{sys}, \textit{inp-trc}) \bullet$$
$$\quad\quad\quad \exists\, \textit{oup-seq} \in \textit{sim}(\textit{sys}, \textit{inp-seq}) \bullet \textit{oup-trc} \in \textit{expand-sequence}(\textit{oup-seq})$$

This says that, for every ELLA text and every input sequence, every corresponding output trace produced by the semantics must be represented by some output sequence produced by the given simulator. According to this, if the semantics produces non-deterministic behaviour, then the simulator must be capable of producing sufficient result sequences to cover all of these output behaviours.

This may be stated in an equivalent way using fewer explicit quantifiers:

$$\textit{sim}\ \textbf{is semantically conservative} \quad \Leftrightarrow$$
$$\quad \forall\, \textit{sys},\ \textit{inp-seq} \bullet \forall\, \textit{inp-trc} \in \textit{expand-sequence}(\textit{inp-seq}) \bullet$$
$$\quad\quad \textit{semantics}(\textit{sys}, \textit{inp-trc}) \subseteq \bigcup \{\ \textit{expand-sequence}(\textit{oup-seq}) \mid \textit{oup-seq} \in \textit{sim}(\textit{sys}, \textit{inp-seq})\ \}$$

However, the above condition simply means that all the traces specified by the semantics are covered by the sequences produced by the simulator. The above condition still allows a simulator to produce *more* traces than necessary — even ones that have no *relation* to those predicted from the semantics. The following condition essentially requires that a simulator should not be misleading in this way.

$$\textit{sim}\ \textbf{is semantically safe} \quad \Leftrightarrow$$
$$\quad \forall\, \textit{sys},\ \textit{inp-seq} \bullet \forall\, \textit{inp-trc} \in \textit{expand-sequence}(\textit{inp-seq}) \bullet$$
$$\quad\quad \forall\, \textit{oup-seq} \in \textit{sim}(\textit{sys}, \textit{inp-seq}) \bullet$$
$$\quad\quad\quad \exists\, \textit{oup-trc} \in \textit{semantics}(\textit{sys}, \textit{inp-trc}) \bullet \textit{oup-trc} \in \textit{expand-sequence}(\textit{oup-seq})$$

This says that, for every ELLA text and every input sequence, every corresponding output sequence produced by the simulator must be *justified* by some output trace produced by the semantics. According to this, the expansion of every output sequence from a simulator must contain some output traces specified by the semantics.

Again, this may be stated in an equivalent way using fewer quantifiers:

$$
\begin{array}{l}
\textit{sim}\ \textbf{is semantically safe} \quad \Leftrightarrow \\
\quad \forall\, \textit{sys},\ \textit{inp-seq} \bullet \forall\, \textit{inp-trc} \in \textit{expand-sequence}(\textit{inp-seq}) \bullet \\
\quad\quad \forall\, \textit{oup-seq} \in \textit{sim}(\textit{sys}, \textit{inp-seq}) \bullet \\
\quad\quad\quad \textit{semantics}(\textit{sys}, \textit{inp-trc}) \cap \textit{expand-sequence}(\textit{oup-seq}) \neq \emptyset
\end{array}
$$

Finally, we say that a *faithful* simulator is one that is both semantically conservative and semantically safe:

$$
\begin{array}{l}
\textit{sim}\ \textbf{is faithful} \quad \Leftrightarrow \\
\quad \textit{sim}\ \textbf{is semantically conservative} \wedge \textit{sim}\ \textbf{is semantically safe}
\end{array}
$$

In practice, a faithful simulator can still be more non-deterministic than the semantics — in the sense that the simulator could always produce '?' for each output. However, such a simulator is clearly not very useful, since it predicts nothing definite about the output!

To say something further about how one simulator can behave 'better' than another, we can introduce a concept of *approximation* between *faithful* simulators as follows:

$$
\begin{array}{l}
\textit{sim}_1 \succeq_{\textit{sim}} \textit{sim}_2 \quad \Leftrightarrow \\
\quad \forall\, \textit{sys},\ \textit{inp-seq} \bullet \\
\quad\quad \forall\, \textit{oup-seq} \in \textit{sim}_2(\textit{sys}, \textit{inp-seq}) \bullet \\
\quad\quad\quad \exists\, \textit{oup-seq}' \in \textit{sim}_1(\textit{sys}, \textit{inp-seq}) \bullet \\
\quad\quad\quad\quad \textit{expand-sequence}(\textit{oup-seq}) \subseteq \textit{expand-sequence}(\textit{oup-seq}')
\end{array}
$$

This says that every output sequence produced by simulator $\textit{sim}_2$ can be matched by an output sequence produced by simulator $\textit{sim}_1$. This relation is a pre-ordering (i.e. reflexive and transitive). Clearly, the simulator that only produces '?' in every trace is the *worst* or least possible simulator, whereas a simulator that mimics the semantics in every respect is clearly the best possible (recall that a trace is simply a query-free sequence of signal vectors). As such, the closer a simulator can reproduce the semantics, the better it will be for predicting behaviour. In practice, this will clearly be traded off against efficiency considerations in that a practical simulator needs to produce its results in a reasonable amount of time.

Finally, we can also consider what happens when a system specified in ELLA possesses deterministic behaviour. Let *sys* be an ELLA description of some system. We can say that *sys* **is deterministic for input** *inp-trc* if and only if

$$\exists\ !\ \textit{oup-trc} \bullet \textit{oup-trc} \in \textit{semantics}(\textit{sys}, \textit{inp-trc})$$

We can also say that *sys* **is totally deterministic** if and only if:

$$\forall\ \textit{inp-trc} \bullet \text{sys } \textbf{is deterministic for input } \textit{inp-trc}$$

We now say what it means for an ideal simulator to *preserve* determinism:

$$\begin{array}{l} \textit{sim}\ \textbf{preserves determinism} \ \Leftrightarrow \\ \quad \textit{sim}\ \textbf{is faithful} \wedge \\ \quad \forall\ \textit{sys}, \textit{inp-trc} \bullet \textit{sys}\ \textbf{is deterministic for input}\ \textit{inp-trc} \ \Rightarrow \\ \qquad \exists\ !\textit{oup-seq} \bullet \textit{oup-seq} \in \textit{sim}(\textit{sys}, \textit{inp-trc}) \wedge \mathsf{card}\, \textit{expand-sequence}(\textit{oup-seq}) = 1 \end{array}$$

Note that the above definition uses input traces rather than the more general input sequences. In effect, the above property says that if the input is fully defined and that the response predicted by the semantics is determinisitic (i.e. unique and fully defined) then the simulation should also be determinisitic. In other words, the ideal simulator should not introduce non-determinism when it doesn't need to.

3.1 The standard ELLA simulator

From the standard product documentation for ELLA, it is fairly clear that the standard ELLA simulator developed by DRA is intended to be *faithful* and that, ideally, it should *preserve determinism* in the above sense.

Unfortunately, the requirement to preserve determinisim is fairly idealistic because it is difficult to ensure in every possible situation without involving further semantic analysis. For example, such a situation can arise when a query value is generated internally within a system and then used by something else. The standard ELLA simulator might then predict that the final output involves '?', even though the exact answer predicted by the semantics is in fact constant and stable.

However, given that the ELLA simulator developed by DRA and the ELLA language have been co-evolving in tandem over a number of years, it is reasonable to assume that the level of design integrity of the simulator and the language are broadly similar, especially since the same design team has been ultimately responsible for both products. Consequently, it is therefore reasonable to expect that both the language and the simulator developed by DRA have reached similar levels of maturity and stability.

In summary, there are many ELLA simulators possible, each implementing an approximation to an ideal semantics. Clearly, some could be better than others — some may even be *erroneous* in that they may either be unsafe or not conservative in some way.

Given the general stability and high integrity of the language design and the fact that the same design team is ultimately responsible for the ELLA simulator, it is reasonable to expect that the ELLA simulator developed by DRA is *faithful* and *preserves determinism* in most practical cases.

4 Semantic structures and relations

In this section, we define some supporting semantic structures needed to present the semantic definition of Core ELLA.

4.1 Assumed structures

We shall assume the following sets as being given:

$\mathbb{Z}$	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 $\mathbb{Z}, \mathbb{N}$ can be assumed.

4.2 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[1].

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.

[1]The behaviour of ELLA strings with respect to '?' is determined by the dynamic semantics.

4.2.1 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.

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 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 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:-

   $$\mathit{tuple\text{-}product}: \mathit{GEN\text{-}TYPE}^+ \rightarrow \mathit{GEN\text{-}TYPE}$$

   $$\mathit{tuple\text{-}product}(\langle ty_1, \cdots, ty_{k+1}\rangle) =$$
   $$\{\langle v_1, \cdots, v_{k+1}\rangle \mid \forall\, i \in \{1,\ldots,k+1\} \bullet\ v_i \in ty_i\}$$

3. $\mathit{TAG\text{-}TYPE} = \mathit{label\text{-}type}(|\mathit{Tag} \times \mathit{ALL\text{-}TYPE}|)$

   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:-

$$Tag = Name$$

$$label\text{-}type : Tag \times ALL\text{-}TYPE \to GEN\text{-}TYPE$$
$$label\text{-}type(tag, aty) = label\text{-}value(|\{tag\} \times aty|)$$

$$TAG\text{-}VALUE ::= \ \texttt{tagval} \ \ tag \ all\text{-}value$$

$$label\text{-}value : Tag \times ALL\text{-}VALUE \to TAG\text{-}VALUE$$
$$label\text{-}value(tag, v) = [\![\, \texttt{tagval} \ \ tag \ v]\!]$$

$$extract\text{-}tag : TAG\text{-}VALUE \to Tag$$
$$extract\text{-}tag[\![\, \texttt{tagval} \ \ tag \ v]\!] = tag$$

$$extract\text{-}value : TAG\text{-}VALUE \to ALL\text{-}VALUE$$
$$extract\text{-}value[\![\, \texttt{tagval} \ \ tag \ v]\!] = v$$

$$extract\text{-}label : TAG\text{-}TYPE \to Tag$$
$$extract\text{-}label(tag\text{-}ty) = \iota \ tag \in Tag \bullet tag \in tag\text{-}set$$
where
$$tag\text{-}set = extract\text{-}tag(|tag\text{-}ty|)$$

$$extract\text{-}type : TAG\text{-}TYPE \to \mathbb{P}(ALL\text{-}VALUE)$$
$$extract\text{-}type(tag\text{-}ty) = extract\text{-}value(|tag\text{-}ty|)$$

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. $GEN\text{-}UNION =$

$$\begin{aligned}
&\{\textstyle\bigcup tag\text{-}type\text{-}set \in GEN\text{-}TYPE \\
&\mid \ tag\text{-}type\text{-}set \in \mathcal{F}(TAG\text{-}TYPE) \bullet \\
&\quad \mathsf{card}\, tag\text{-}type\text{-}set = \mathsf{card}\, extract\text{-}label(|tag\text{-}type\text{-}set|) \wedge \\
&\quad tag\text{-}type\text{-}set \neq \emptyset \\
&\}
\end{aligned}$$

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 Kernal ELLA type, `typevoid`.

7. $ATOMIC = INT\text{-}RANGE \cup CHAR\text{-}SET \cup STRING$

   The atomic types added within *ALL-TYPE* consist of compact integer intervals, character subsets and sets of bounded character strings. 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.

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).

4.3 Automata theoretic semantics

We give a semantics for Core ELLA, a well-defined subset of the ELLA language, in terms of *input/output automata* (or *IO* automata) in a similar way to our earlier semantics work[MDED91, BGL+91]. Each ELLA function is mapped by the semantics into an IO automaton. Each of these automata defines an *IO language* as a set of behavioural traces. Accordingly, we can say that two ELLA functions are *semantically equivalent* whenever their corresponding IO automata are *trace equivalent* (i.e they *accept* the same IO language).

An *IO automaton*, *A*, is specified by the following data:-

- a finite set of states

- input signature map of type *Signature*
- output signature map of type *Signature*
- a set of initial states
- a finite set of transitions.

The input and output signature for automaton A determines the structure of the input and output mappings labelling the transitions of A. Each transition has the form:-

$$(s_1 \xrightarrow{m} s_2)$$

for states s_1, s_2 is labelled by a single binding, m, for the input and output signals, with domain given by the union of the domains of the input and output signatures. The type of each signal value in m is determined by the corresponding type in either the input or output signature.

More formally, we give the following definitions:

$$\begin{aligned}
&\mathit{Signature} = \mathit{Name} \xrightarrow{\text{map}} \mathit{GEN\text{-}TYPE} \\
&\mathit{Binding} = \mathit{Name} \xrightarrow{\text{map}} \mathit{GEN\text{-}VALUE}
\end{aligned}$$

The set *Signature* defines the set of type assignments, and the set *Binding* defines the set of value assignments.

$$\mathit{STATE} = \mathbb{N} \cup \mathit{GEN\text{-}VALUE} \cup \mathit{STATE}^+$$

We have choosen to give a concrete definition for the set of all possible automaton states, *STATE*. This is used to provide structured objects to provide suitable 'labels' serving to denote states. A more abstract definition is certainly possible — by characterising the set of all states algebraiclly in terms of access and construction functions and also inspection predicates. However, the concrete definition has the virtues of being concise, simple and direct.

$$TRANS = \{(s_1 \xrightarrow{m} s_2) \mid s_1, s_2 \in STATE, m \in Binding\}$$

$GEN\text{-}IOA ::=$ `ioa` $states\ input\text{-}sig\ output\text{-}sig\ initials\ transitions$

where

$\mathsf{dom}\, input\text{-}sig \cap \mathsf{dom}\, output\text{-}sig = \emptyset\ \wedge\ initials \subseteq states\ \wedge$

$(\forall\ (s_1 \xrightarrow{m} s_2) \in transitions \bullet$

$\{s_1, s_2\} \subseteq states \wedge m \in MapType(input\text{-}sig \cup output\text{-}sig)$

$)$

$$States = \mathcal{F}(STATE)$$
$$Initials = States$$
$$Input\text{-}Sig = Signature$$
$$Output\text{-}Sig = Signature$$
$$Transitions = \mathcal{F}(TRANS)$$

The *MapType* operator produces the set of all value bindings matching a given type signature, $sig : Signature$:

$$MapType : Signature \to \mathbb{P}(Binding)$$
$$MapType(sig) = \{m \in Binding \mid \mathsf{dom}\, m = \mathsf{dom}\, sig \wedge \forall\ nm \in \mathsf{dom}\, m \bullet m(nm) \in sig(nm)\}$$

The type operator *MapType* conveniently provides a simple form of dependent typing.

The input/output signature operators are formally specified by:-

$$\Sigma_I : GEN\text{-}IOA \to Signature$$
$$\Sigma_I[\![\ \texttt{ioa}\ \ st\ i\text{-}sig\ o\text{-}sig\ in\text{-}st\ tns]\!] = i\text{-}sig$$

$$\Sigma_O : GEN\text{-}IOA \to Signature$$
$$\Sigma_O[\![\ \texttt{ioa}\ \ st\ i\text{-}sig\ o\text{-}sig\ in\text{-}st\ tns]\!] = o\text{-}sig$$

Since the input/output signatures of an IOA must be disjoint, we may define the *joint signature*, Σ_{IO}, as follows:

$$\Sigma_{IO} :\ GEN\text{-}IOA \to Signature$$
$$\Sigma_{IO}(A) = \Sigma_I(A) \cup \Sigma_O(A)$$

We will occasionally need to extract semantic type and resulting value information from a semantic object; this is provided by the functions *out-type* and *out-values* respectively.

$$out\text{-}type : GEN\text{-}IOA \rightarrow GEN\text{-}TYPE$$
$$out\text{-}type(A) = \bigcup\{ty \in GEN\text{-}TYPE \mid ty \in \mathsf{rng}\,\Sigma_O(A)\}$$

$$out\text{-}values : GEN\text{-}IOA \rightarrow \mathbb{P}(GEN\text{-}VALUE)$$
$$out\text{-}values(A) = \bigcup\{\mathsf{rng}\,out\text{-}map \mid out\text{-}map \in map\text{-}set\}$$
where
$$map\text{-}set = \{\ (out\text{-}sig \lhd m) \mid (s_1 \xrightarrow{m} s_2) \in Transitions(A)\ \}$$
$$out\text{-}sig = \mathsf{dom}\,\Sigma_O(A)$$

It turns out that these functions are mainly used in cases where there is precisely a single output for the given automaton. However, the above general definitions for them makes it unnecessary to depend upon the specific *names* used for the output.

4.3.1 Representation issues for automata and transitions

In our previous semantic descriptions[MDED91, BGL$^+$91], we have generally separated "behaviour objects" (e.g. automata) from "signal objects" (e.g. values in transition mappings).

Defined ELLA functions are semantically denoted by IO automata which have a standard form of input/output signature:

- positive numbers for each corresponding input argument
- 0 (zero) naming the single output value

In our earlier work, the transition mapping *m* would only have been weakly constrained so that its domain was merely contained within the RHS, rather than being equal to it as the above requires. In terms of enumerative techniques of verification, this allowed the necessary flexibility to permit the representation of the transition relation to be reduced by eliminating redundency. However, the complexity of data typing permitted within Core ELLA rules out a straightforward enumerative approach to verification and so this extra flexibility is unnecessary. Other techniques can be used to 'bundle' the transitions into relational groupings (e.g. predicate/action rules).

4.3.2 Automata composition primitives

Automata terms denote the behaviour of a system and basically consist of terms built according to the following (operator) signature:-

$(\_ \parallel \_)$	:	$GEN\text{-}IOA \times GEN\text{-}IOA \rightarrow GEN\text{-}IOA$	causal product
$(\_[\_])$	:	$GEN\text{-}IOA \times Renaming \rightarrow GEN\text{-}IOA$	renaming
$(\_\backslash\_)$	:	$GEN\text{-}IOA \times \mathcal{F}(Name) \rightarrow GEN\text{-}IOA$	hiding

where we have that:

$$Renaming = Name \xleftrightarrow{\textsf{map}} Name$$

i.e. *Renaming* are the finite *injective* mappings from *Name* to *Name* — note that such mappings can always be 'inverted'.

The generic inversion operator, $\textit{inv-map}_{[A,B]}$ for these injective partial mappings is defined by:-

$$\textit{inv-map}_{[A,B]} : (A \xleftrightarrow{\textsf{map}} B) \to (B \xleftrightarrow{\textsf{map}} A)$$
$$\textit{inv-map}_{[A,B]}(m) = \{m(a) \mapsto a \mid a \in \mathsf{dom}\, m\}$$

The above operators provide the basic automaton connectives. Other primitives will be specified by presenting their transition relations in set-theoretical terms. It is also possible to build up an algebraic characterisation of IO automata, in which it should be possible to give equational definitions of these primitives. Such definitions are given in terms of the IO languages accepted by the primitives.

The inferential semantics of *causal product, hiding* and *renaming* are given below:

$$(\_ \;||\; \_) : \textit{GEN-IOA}^2 \to \textit{GEN-IOA}$$
$$(A \;||\; B) = [\![\texttt{ioa}\;\; \textit{states i-sig o-sig inits trans-set}]\!]$$

where

$$\textit{states} = \textit{States}(A) \times \textit{States}(B)$$
$$\textit{i-sig} = \Sigma_I(A) \cup \Sigma_I(B) - \Sigma_O(C)$$
$$\textit{o-sig} = \Sigma_O(A) \cup \Sigma_O(B)$$
$$\textit{inits} = \textit{Inits}(A) \times \textit{Inits}(B)$$

$$\boxed{\textsf{causal product}}\; \frac{\begin{array}{ll}\text{(a)} & (x_1 \xrightarrow{m_1} x_2) \in \textit{Trans}(A) \\ \text{(b)} & (y_1 \xrightarrow{m_2} y_2) \in \textit{Trans}(B)\end{array}}{\langle x_1, y_1\rangle \xrightarrow{m_1 \,\cup\, m_2} \langle x_2, y_2\rangle \in \textit{trans-set}}$$

pre

$\Sigma_{IO}(A)$ **is compatible with** $\Sigma_{IO}(B)$

$(\_\backslash\_) : \mathit{GEN\text{-}IOA} \times \mathcal{F}(\mathit{Name}) \to \mathit{GEN\text{-}IOA}$

$(A \backslash S) = [\![\ \texttt{ioa}\ \ \mathit{states}\ \mathit{i\text{-}sig}\ \mathit{o\text{-}sig}\ \mathit{inits}\ \mathit{trans\text{-}set}]\!]$

where

$\mathit{states} = \mathit{States}(A)$

$\mathit{i\text{-}sig} = S \ntriangleleft \Sigma_I(A)$

$\mathit{o\text{-}sig} = S \ntriangleleft \Sigma_O(A)$

$\mathit{inits} = \mathit{Inits}(A)$

$$\boxed{\text{hiding}}\ \frac{(x_1 \xrightarrow{m} x_2) \in \mathit{Trans}(A)}{(x_1 \xrightarrow{S \ntriangleleft m} x_2) \in \mathit{trans\text{-}set}}$$

$(\_[\_]) : \mathit{GEN\text{-}IOA} \times \mathit{Renaming} \to \mathit{GEN\text{-}IOA}$

$(A[\mathit{ren}]) = [\![\ \texttt{ioa}\ \ \mathit{states}\ \mathit{i\text{-}sig}\ \mathit{o\text{-}sig}\ \mathit{inits}\ \mathit{trans\text{-}set}]\!]$

where

$\mathit{ren\text{-}inv} = \mathit{inv\text{-}map}_{[\mathit{Name},\mathit{Name}]}(\mathit{ren})$

$\mathit{ren\text{-}map} = \{\mathit{nm} \mapsto \mathit{nm} \mid \mathit{nm} \in \Sigma_{IO}(A)\}\ \dagger\ \mathit{ren\text{-}inv}$

$\mathit{states} = \mathit{States}(A)$

$\mathit{i\text{-}sig} = \Sigma_I(A)\ \circ\ \mathit{ren\text{-}map}$

$\mathit{o\text{-}sig} = \Sigma_O(A)\ \circ\ \mathit{ren\text{-}map}$

$\mathit{inits} = \mathit{Inits}(A)$

$$\boxed{\text{renaming}}\ \frac{(x_1 \xrightarrow{m} x_2) \in \mathit{Trans}(A)}{(x_1 \xrightarrow{m\ \circ\ \mathit{ren\text{-}map}} x_2) \in \mathit{trans\text{-}set}}$$

pre

$\Sigma_{IO}(A)$ **is compatible with** $(\Sigma_{IO}(A)\ \circ\ \mathit{ren\text{-}inv})$

The compatibility relation used above is specified as follows:

$(\_$ **is compatible with** $\_)\ \subseteq \mathit{Signature}^2$

sig_1 **is compatible with** $\mathit{sig}_2\ \Leftrightarrow$

$\forall\, x \in (\mathsf{dom}\, \mathit{sig}_1 \cap \mathsf{dom}\, \mathit{sig}_2) \bullet \mathit{sig}_1(x) = \mathit{sig}_2(x)$

The above slightly extends the denotational semantics of the above operators defined in earlier work[MDED91, BGL+91].

4.3.3 CCS and IO automata

We note that the CCS notion of *process* corresponds to 'automaton in a particular state'. Observe that the automaton combinators ($||$, $\times$, $\_\backslash\_$, $\_[\_]$) broadly correspond to Milner's static combinators for CCS processes. A set of defining CCS equations for particular processes corresponds in the *GEN-IOA* setting to specifications of automata in terms of their transition relations, together with initial state information. Finally, we note that in [Mil90], Milner uses the term 'Restriction' to mean what we have termed 'hiding', although we use the same symbolic notation.

4.4 IO Traces and Languages, Trace Equivalence and Containment

The notions of *Trace*, *IO language*, *Trace Equivalence* and *Trace Containment* are straightforward generalisations of the definitions given in [MDED91, BGL+91], extended to include the semantic concept of typing, as required by ELLA.

A *(typed) IO trace* is a finite sequence of mappings, each of whose elements conforms to an appropriate input/output signature. An *IO language*, L is a *prefix closed* collection of IO traces (i.e $(tr_1 \frown tr_2) \in L \;\Rightarrow\; tr_1 \in L$). Let A be an IO automaton, and let $Lang_{IO}(A)$ stand for the IO language accepted by A. Automata A, B are said to be *Trace Equivalent* whenever they both *accept* the same IO language (i.e. $Lang_{IO}(A) = Lang_{IO}(B)$). Finally, we say that the behaviours of A is *trace contained* within the behaviour of B whenever the respective IO languages are contained (i.e. $Lang_{IO}(A) \subseteq Lang_{IO}(B)$).

5 Semantic definition

In this major section, we present a Core ELLA semantics in the horizontal form discussed earlier. The additional notation used for presenting the semantics is given in Appendix B.

A couple of appendices (Appendices C and D) re-presents this information, but in a vertical organisation to enable the static and dynamic semantics to be independently inspected as a coherent whole.

5.1 Binding Environments

To present this semantics of Core ELLA, we shall need to use a couple of environment structures whose purpose is to accumulate both static and dynamic information concerning the type and function object bindings.

5.1.1 Dynamic Environments

A *dynamic environment* is specified in terms of *Environment* objects and is used to hold the bindings of semantic objects that have been determined so far:

$$\begin{aligned} \mathit{Environment} &= \mathit{Name} \xrightarrow{\textsf{map}} \mathit{ENV\text{-}DATA} \\ \mathit{ENV\text{-}DATA} &= (\mathit{TYPE\text{-}DATA} \cup \mathit{OBJ\text{-}DATA}) \end{aligned}$$

These semantic *Environment* objects contain named denotations for either objects or types. We shall conveniently assume that the name space has enough structure to ensure that this simple scheme is sufficient (see remarks below).

The semantic type information is determined as follows:

$$\mathit{TYPE\text{-}DATA} = (\mathit{GEN\text{-}TYPE} \cup \mathit{Tag}^{+})$$

Note that (non-empty) sequences of *Tag* can be used within environments, in order to record the ordering information given by definitions of enumerated types.

Information concerning semantic objects is structured as follows:

$$\begin{aligned} \mathit{OBJ\text{-}DATA} &::= \mathit{gen\text{-}ioa} \mid \mathit{blocksig} \\ \mathit{BlockSig} &::= \mathit{makesig} \mid \mathit{letsig} \\ \mathit{MakeSig} &::= \texttt{make-signal}\ \mathit{gen\text{-}ioa}_1 \mathit{gen\text{-}ioa}_2 \\ \mathit{LetSig} &::= \texttt{let-signal}\ \mathit{gen\text{-}ioa} \end{aligned}$$

Semantic objects can either be an ELLA function denotation (as a general IOA), or it is information concerning an ELLA signal, defined locally within a block. In order to semantically capture the `MAKE` / `JOIN` approach used to define circuits, it will be necessary to distinguish between signals introduced by `MAKE` and those introduced either by `LET` or as formal parameters. This is mainly because of the dual role that `MAKE` signals play in naming *both* the object to be **joined** to as well as a particular point in a circuit.

We generally assume that $env \in Environment$.

5.1.2 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. Scoping of names is considered to be an integral part of the well-formedness properties of the language.

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

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 $st\text{-}env \in StaticEnvironment$.

5.2 Declaration information

Structure:

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

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:

$$\mathit{type\text{-}env} : \mathit{TypeDecl}^* \rightarrow \mathit{StaticEnvironment}$$

$$\mathit{type\text{-}env}[\![\mathit{td}, \mathit{td}_1,\ \cdots\ , \mathit{td}_k]\!] = \{\mathit{nm} \mapsto \mathit{td}\} \cup \mathit{st\text{-}env}$$

where

$$\mathit{nm} = \mathit{type\text{-}decl\text{-}name}(\mathit{td}),$$

$$\mathit{st\text{-}env} = \mathit{type\text{-}env}[\![\mathit{td}_1,\ \cdots\ , \mathit{td}_k]\!]$$

$$\mathit{type\text{-}env}[\![]\!] = \emptyset$$

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

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

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

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

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

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

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

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

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

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

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

$NewTypeParts = TagType \cup IntType \cup CharType$

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

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

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

$newtype\text{-}parts\text{-}of : NewType \rightarrow \mathcal{F}(NewTypeParts)$

$newtype\text{-}parts\text{-}of[\![tagty_1 \ = \ \cdots \ = \ tagty_{k+1}]\!] = \{tagty_1,\ \cdots,\ tagty_{k+1}\}$

$newtype\text{-}parts\text{-}of[\![nty]\!] = \{nty\}$

where

$nty \in (IntType \cup CharType)$

$vartype\text{-}of : BlockSig \rightarrow Type$

$vartype\text{-}of[\![\ \texttt{let-type}\ obj\text{-}ty]\!] = obj\text{-}ty$

$vartype\text{-}of[\![\ \texttt{make-type}\ arg\text{-}tyobj\text{-}ty]\!] = obj\text{-}ty$

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

$join\text{-}to\text{-}names : Step^* \rightarrow \mathcal{F}(Name)$

$join\text{-}to\text{-}names[\![st_1,\ \cdots\ ,\ st_n]\!] = \{join\text{-}to\text{-}name(st_i) \mid i \in \{1,\ldots,n\}\}$

$join\text{-}to\text{-}name : Step \rightarrow Name$

$join\text{-}to\text{-}name(step) = \emptyset$

where

$step \in (Step - Make)$

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

The following rules define various analysis predicates:

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

$$\forall\ k \in \mathbb{N}, chr_1, \cdots, chr_{k+1} \in \textit{Char}, tag \in \textit{Tag}, \textit{new-ty} \in \textit{NewType},$$

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

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

$$\forall\ i, j \in \mathbb{Z}, tag \in \textit{Tag}, \textit{new-ty} \in \textit{NewType},$$

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

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

$$\forall\ n_1 \in \mathbb{N}, ty_1 \in \textit{Type}, tag \in \textit{Tag}, \textit{new-ty} \in \textit{NewType},$$

$$\dfrac{\begin{array}{ll} \text{(a)} & tynm \in \mathsf{dom}\, \textit{st-env} \\ \text{(b)} & \textit{st-env}(tynm) = [\![tynm = \ \texttt{STRING}\ [n_1] ty_1]\!] \end{array}}{\textit{st-env} \vdash_{\textit{is-string-type}} ty}\ \boxed{\text{is-string-type}}$$

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

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

$$\textit{range-set}[\![tagty_1 \ \underline{=}\ \cdots \ \underline{=}\ tagty_{k+1}]\!] = \emptyset$$

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

$$\textit{range-set}[\![tag('chr_1 \ \underline{=}\ \cdots \ \underline{=}\ 'chr_k)]\!] = \{chr_1, \cdots, chr_k\}$$

The following function is used to extract set values from either a given type or from a finite sequence of tags/characters representing the ordering upon an enumerated type:

$TAG\text{-}CHAR = \{label\text{-}type(tag, chr\text{-}type) \in TAG\text{-}TYPE \mid tag \in Tag \wedge chr\text{-}type \in CHAR\text{-}SET\}$

$extract\text{-}set : (GEN\text{-}TYPE \cup Tag^* \cup TAG\text{-}CHAR^*) \to GEN\text{-}TYPE$

$extract\text{-}set(s) = s$

where

$s \in (GEN\text{-}TYPE - Tag^* - TAG\text{-}CHAR^*)$

$extract\text{-}set\ \langle tag_1,\ \cdots\ ,\ tag_{k+1}\rangle = \{tag_1,\ \cdots\ ,\ tag_{k+1}\}$

$extract\text{-}set\ \langle label\text{-}value(tag, chr_1),\ \cdots\ ,\ label\text{-}value(tag, chr_{k+1})\rangle =$
$\{label\text{-}value(tag, chr_i) \mid i \in \{1, \ldots, k+1\}\}$

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

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

is-const-value : Enumerated

$$\frac{st\text{-}env \vdash_{Enumerated} enum}{st\text{-}env \vdash_{is\text{-}const\text{-}value} enum}$$

is-const-value : Row

$$\frac{\forall\ i \in \{1, \ldots, k+1\} \bullet st\text{-}env \vdash_{is\text{-}const\text{-}value} cnst_i}{st\text{-}env \vdash_{is\text{-}const\text{-}value} [\![cnst_1,\ \cdots\ , cnst_{k+1}]\!]}$$

is-const-value : String

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

is-const-value : Query

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

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

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

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

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

5.2.1 ELLA type equivalence

The following rules define the type equivalence relation, for use by the static semantics. Notice that this relation is unnecessary in the dynamic semantics since the complete mathematical denotation of any type expressions is available.

$$(\_ \vdash_{\textit{Type-Equality}} \_ \;\equiv\; \_) \subseteq (\textit{StaticEnvironment} \times \textit{Type} \times \textit{Type})$$

$$\boxed{\textsf{Type-Equality : Reflexivity}}\;\frac{\textit{st-env} \vdash_{\textit{Type}} \textit{ty}}{\textit{st-env} \vdash_{\textit{Type-Equality}} \textit{ty} \;\equiv\; \textit{ty}}$$

$$\boxed{\textsf{Type-Equality : Transitivity}}\;\frac{\begin{array}{ll}\text{(a)} & \textit{st-env} \vdash_{\textit{Type-Equality}} \textit{tya} \;\equiv\; \textit{tyb}\\ \text{(b)} & \textit{st-env} \vdash_{\textit{Type-Equality}} \textit{tyb} \;\equiv\; \textit{tyc}\end{array}}{\textit{st-env} \vdash_{\textit{Type-Equality}} \textit{tya} \;\equiv\; \textit{tyc}}$$

$$\boxed{\textsf{Type-Equality : Symmetry}}\;\frac{\textit{st-env} \vdash_{\textit{Type-Equality}} \textit{tyb} \;\equiv\; \textit{tya}}{\textit{st-env} \vdash_{\textit{Type-Equality}} \textit{tya} \;\equiv\; \textit{tyb}}$$

$$\boxed{\textsf{Type-Equality : TypeName}}\;\frac{\begin{array}{ll}\text{(a)} & \textit{tynm} \in \textsf{dom}\,\textit{st-env}\\ \text{(b)} & \textit{st-env}(\textit{tynm}) = [\![\textit{tynm} = \textit{rhs-ty}]\!]\\ \text{(c)} & \textit{st-env} \vdash_{\textit{Type-Equality}} \textit{rhs-ty} \;\equiv\; \textit{ty}\end{array}}{\textit{st-env} \vdash_{\textit{Type-Equality}} \textit{tynm} \;\equiv\; \textit{ty}}$$

$$\boxed{\textsf{Type-Equality : TupleType}}\;\frac{\begin{array}{ll}\text{(a)} & n \leq 1\\ \text{(b)} & \forall\, i \in \{1,\ldots,n\} \bullet \textit{st-env} \vdash_{\textit{Type-Equality}} \textit{tya}_i \;\equiv\; \textit{tyb}_i\end{array}}{\textit{st-env} \vdash_{\textit{Type-Equality}} [\![(\textit{tya}_1,\ \cdots\ ,\ \textit{tya}_{n+1})]\!] \;\equiv\; [\![(\textit{tyb}_1,\ \cdots\ ,\ \textit{tyb}_{n+1})]\!]}$$

$$\boxed{\textsf{Type-Equality : PowerType}}\;\frac{\begin{array}{ll}\text{(a)} & \textit{st-env} \vdash_{\textit{Type-Equality}} [\![(\textit{ty}_1,\ \cdots\ ,\ \textit{ty}_n)]\!] \equiv \textit{tyb}\\ \text{(b)} & \forall\, i \in \{1,\ldots,n\} \bullet \textit{ty}_i = \textit{tya}\end{array}}{\textit{st-env} \vdash_{\textit{Type-Equality}} [\![[n]\textit{tya}]\!] \;\equiv\; \textit{tyb}}$$

$$\boxed{\textsf{Type-Equality : StringType}}\;\frac{\textit{st-env} \vdash_{\textit{Type-Equality}} \textit{tya} \;\equiv\; \textit{tyb}}{\textit{st-env} \vdash_{\textit{Type-Equality}} [\![\ \texttt{STRING}\ [n]\textit{tya}]\!] \;\equiv\; [\![\ \texttt{STRING}\ [n]\textit{tyb}]\!]}$$

5.2.2 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:

$$\textit{free-signals} : (\textit{Unit} \cup \textit{Step} \cup \textit{Caseitem} \cup \textit{SignalDecl}) \rightarrow \mathcal{F}(\textit{Name})$$

$un : Unit$	$\textit{free-signals}(un)$
$(\,)$	$\emptyset$
$enum\ \&\ unit$	$\textit{free-signals}(unit)$
STRING $[nat]unit$	$\textit{free-signals}(unit)$
$[\ nat\ ]\ unit$	$\textit{free-signals}(unit)$
$?\ type$	$\emptyset$
$unit\ [\ nat\ ]$	$\textit{free-signals}(unit)$
$unit_1\ [[\ unit_2\ ]]$	$\textit{free-signals}(unit_1) \cup \textit{free-signals}(unit_2)$
REPLACE $(unit_1, unit_2, unit_3)$	$\textit{free-signals}(unit_1) \cup \textit{free-signals}(unit_2)$ $\cup\, \textit{free-signals}(unit_3)$
$unit\ [\ nat_1\ \underline{..}\ nat_2\ ]$	$\textit{free-signals}(unit)$
$unit_1$ CONC $unit_2$	$\textit{free-signals}(unit_1) \cup \textit{free-signals}(unit_2)$
$(unit_1\ ,\ \ldots\ ,\ unit_{k+1})$	$\bigcup\{\textit{free-signals}(un_i) \mid i \in \{1, \ldots, k+1\}\}$
$unit\ //\ enum$	$\textit{free-signals}(unit)$
$signal$	$\{signal\}$
$fnname\ (\ unit\ )$	$\textit{free-signals}(unit)$
BEGIN $step_1 \ldots step_{k+1}$ OUTPUT $unit$ END	$(\ \textit{free-signals}(unit) \cup$ $\bigcup\{\textit{free-signals}(step_i) \mid i \in \{1, \ldots, k+1\}\}$ $) - \textit{local-signals}[\![step_1 \ldots step_{k+1}]\!]$
CASE $unit_1$ OF $caseitem_1\ ,\ \ldots\ ,\ caseitem_{k+1}$ ELSE $unit_2$ ESAC	$\textit{free-signals}(unit_1) \cup \textit{free-signals}(unit_2) \cup$ $\bigcup\{\textit{free-signals}(caseitem_i) \mid i \in \{1, \ldots, k+1\}\}$

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

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

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

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

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

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

5.2.3 New Types and Tag Types

Structure:

$$
\begin{array}{|rcl|}
\hline
\mathit{NewType} & ::= & \mathit{assoctype} \mid \mathit{inttype} \mid \mathit{chartype} \\
\mathit{AssocType} & ::= & (\mathit{tagtype}_1 \underline{|} \ldots \underline{|} \mathit{tagtype}_{k+1}) \\
\mathit{IntType} & ::= & \mathit{tag}/(\mathit{int}_1 \mathrel{\underline{..}} \mathit{int}_2) \\
\mathit{CharType} & ::= & \mathit{tag}(\,'\mathit{char}_1 \underline{|} \ldots \underline{|}\,'\mathit{char}_{k+1}) \\
\mathit{TagType} & ::= & \mathit{tag} \mid \mathit{tag}\ \&\ \mathit{type} \\
\hline
\end{array}
$$

Static semantics rules:

$$
\boxed{\textsf{wf-NewType : AssocType}}\quad
\frac{\begin{array}{ll}
\text{(a)} & \mathit{tag\text{-}set} = \mathit{tags\text{-}of\text{-}newtype}[\![(\mathit{tagty}_1 \underline{|} \cdots \underline{|} \mathit{tagty}_{k+1})]\!] \\
\text{(b)} & \mathsf{card}\,\mathit{tag\text{-}set} = k+1 \\
\text{(c)} & \forall\, i \in \{1,\ldots,k+1\} \bullet \mathit{st\text{-}env} \vdash_{\mathit{TagType}} \mathit{tagty}_i
\end{array}}{\mathit{st\text{-}env} \vdash_{\mathit{NewType}} (\mathit{tagty}_1 \underline{|} \ldots \underline{|} \mathit{tagty}_{k+1})}
$$

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

$$
\boxed{\textsf{wf-NewType : CharType}}\quad
\frac{\begin{array}{ll}
\text{(a)} & \mathit{tag} \notin \mathit{tags\text{-}of\text{-}env}(\mathit{st\text{-}env}) \\
\text{(b)} & \mathit{char\text{-}set} = \mathit{range\text{-}set}[\![\mathit{tag}(\,'\mathit{char}_1 \underline{|} \ldots \underline{|}\,'\mathit{char}_{k+1})]\!] \\
\text{(c)} & \mathsf{card}\,\mathit{char\text{-}set} = k+1
\end{array}}{\mathit{st\text{-}env} \vdash_{\mathit{NewType}} \mathit{tag}(\,'\mathit{char}_1 \underline{|} \ldots \underline{|}\,'\mathit{char}_{k+1})}
$$

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

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

Dynamic semantics equations:

$$\textit{M-NewType}[\![(\textit{tagty}_1 \underline{|} \ldots \underline{|} \textit{tagty}_{k+1})]\!](\textit{env}) = \langle \textit{tag}_1, \ \cdots \ , \textit{tag}_{k+1} \rangle$$

where

$\forall\ i \in \{1, \ldots, k+1\} \bullet$

$\textit{tagty}_i = \textit{tag}_i \wedge \textit{tag}_i \in \textit{Tag}$

$$\textit{M-NewType}[\![(\textit{tagty}_1 \underline{|} \ldots \underline{|} \textit{tagty}_{k+1})]\!](\textit{env}) =$$
$$\bigcup \{\textit{ty}_i \in \textit{GEN-TYPE} \mid i \in \{1, \ldots, k+1\}\}$$

where

$\forall\ i \in \{1, \ldots, k+1\} \bullet$

$\textit{ty}_i = \textit{M-TagType}(\textit{tagty}_i)(\textit{env})$

$\exists\ i \in \{1, \ldots, k+1\} \bullet$

$\textit{tagty}_i = [\![\textit{tag}\ \&\ \textit{ty}]\!]$

$$\textit{M-NewType}[\![\textit{tag}/(i \underline{..}\, j)]\!](\textit{env}) = \textit{label-type}(\textit{tag}, \{i, \ldots, j\})$$

$$\textit{M-NewType}[\![\textit{tag}(\ '\,\textit{char}_1 \underline{|} \ldots \underline{|}\ '\,\textit{char}_{k+1})]\!](\textit{env}) =$$
$$\langle \textit{label-value}(\textit{tag}, \textit{char}_1), \ \cdots \ , \textit{label-value}(\textit{tag}, \textit{char}_{k+1}) \rangle$$

$$\textit{M-TagType}[\![\textit{tag}\ \&\ \textit{ty}]\!](\textit{env}) = \textit{label-type}(\textit{tag}, \textit{M-Type}(\textit{ty})(\textit{env}))$$

$$\textit{M-TagType}[\![\textit{tag}]\!](\textit{env}) = \{\textit{tag}\}$$

5.3 Types

Structure:

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

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

$$
\begin{array}{l}
(\_ \vdash_{\textit{Type}} \_) \subseteq (\textit{StaticEnvironment} \times \textit{Type}) \\
\textit{M-Type} : \textit{Type} \times \textit{Environment} \rightarrow \textit{GEN-TYPE}
\end{array}
$$

Static semantics rules:

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

$$
\boxed{\text{wf-Type : TupleType}}\ \frac{\forall\, i \in \{1, \ldots, k+1\} \cdot \textit{st-env} \vdash_{\textit{Type}} \textit{ty}_i}{\textit{st-env} \vdash_{\textit{Type}} (\textit{ty}_1, \ \cdots \ , \textit{ty}_{k+1})}
$$

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

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

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

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

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

Dynamic semantics equations:

$$\textit{M-Type}[\![\textit{tynm}]\!](\textit{env}) = \textit{extract-set}(\textit{env}(\textit{tynm}))$$

$$\textit{M-Type}[\![(ty_1)]\!](\textit{env}) = \textit{M-Type}[\![ty_1]\!](\textit{env})$$

$$\textit{M-Type}[\![(ty_1, \cdots, ty_{k+1})]\!](\textit{env}) = \textit{tuple-product}\ \langle \textit{type}_1, \cdots, \textit{type}_{k+1} \rangle$$

where

$$\forall\ i \in \{1, \ldots, k+1\} \bullet \textit{type}_i = \textit{M-Type}[\![ty_i]\!](\textit{env})$$

$$k \geq 1$$

$$\textit{M-Type}[\![[n]ty]\!](\textit{env}) = \textit{tuple-product}\ \langle \textit{type}_1, \cdots, \textit{type}_n \rangle$$

where

$$\forall\ i \in \{1, \ldots, n\} \bullet \textit{type}_i = \textit{M-Type}[\![ty]\!](\textit{env})$$

$$\textit{M-Type}[\![\ \texttt{STRING}\ [n]ty]\!](\textit{env}) =$$

$$\textit{label-type}(\textit{tag}, \textit{tuple-product}(\langle \textit{type}_1, \cdots, \textit{type}_n \rangle))$$

where

$$\exists\ !\textit{chr-type} \subseteq \textit{Char} \bullet$$

$$\textit{env}(ty) = \textit{label-type}(\textit{tag}, \textit{chr-type}) \wedge$$

$$\forall\ i \in \{1, \ldots, n\} \bullet\ \textit{type}_i = \textit{chr-type}$$

$$\textit{M-Type}[\![(\ )]\!](\textit{env}) = \{\ \texttt{unitvoid}\ \}$$

Note that for a tuple-type containing a single element, then the sequencing brackets are ignored and the semantics for the elemental type is returned.

5.4 Type declarations

Structure:

$$\begin{array}{rcl} \mathit{TypeDecl} & ::= & \mathit{typenaming} \mid \mathit{typeformer} \\ \mathit{TypeNaming} & ::= & \texttt{TYPE}\ \mathit{typename} = \mathit{type} \\ \mathit{TypeFormer} & ::= & \texttt{TYPE}\ \mathit{typename} = \texttt{NEW}\ \mathit{newtype} \end{array}$$

The *type declaration* introduces a name for a type structure. The first form above introduces a synonym for an type that is already formed, whereas the second introduces a freshly named primitive structure.

Static semantics rules:

Let $ty \in \mathit{Type}$:

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

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

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

A type declaration is well-formed precisely when:

1. The type name being introduced has not already been defined.

2. Synonym types must already be well-formed. Any aliasing of names is resolved.

Dynamic semantics equations:

$$\textit{M-Typedecl}[\![\ \texttt{TYPE}\ \textit{tynm} = \textit{ty}]\!](\textit{env}) = \textit{env} \cup \{\textit{tynm} \mapsto \textit{env}(\textit{ty})\}$$

where

$$\textit{ty} \in \textit{TypeName}$$

$$\textit{M-Typedecl}[\![\ \texttt{TYPE}\ \textit{tynm} = \textit{ty}]\!](\textit{env}) = \textit{env} \cup \{\textit{tynm} \mapsto \textit{M-Type}[\![\textit{ty}]\!](\textit{env})\}$$

where

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

$$\textit{M-Typedecl}[\![\ \texttt{TYPE}\ \textit{tynm} = \ \texttt{NEW}\ \textit{nty}]\!](\textit{env}) = \textit{env} \cup \{\textit{tynm} \mapsto \textit{type}\}$$

where

$$\textit{type} = \textit{M-Newtype}[\![\textit{nty}]\!](\textit{env})$$

The semantics of each type declaration is to add a name binding for the declared type, after appropriately elaborating any newly-tagged types.

5.4.1 Enumerated

Structure:

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

Enumerated simply returns a constant value v of the appropriate type. A basic automaton 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 $tag/taglit$, a tagged character $tag'\ chr$, or a tagged string $tag\ ''\ ch_1\ \ldots\ ch_{k+1}\ ''$.

Static semantics rules:

wf-Enumerated: Tag

$$
\frac{\begin{array}{ll}\text{(a)} & tyn \in \mathsf{dom}\, st\text{-}env \wedge newtype\text{-}of(st\text{-}env(tyn)) \subset Tag \\ \text{(b)} & tag \in newtype\text{-}of(st\text{-}env(tyn))\end{array}}{st\text{-}env \vdash_{Enumerated} [\![tag]\!] \text{ has type } tyn}
$$

wf-Enumerated: EnumInt

$$
\frac{\begin{array}{ll}\text{(a)} & tyn \in \mathsf{dom}\, st\text{-}env \wedge [\![tag/(lb\underline{\ldots}ub)]\!] \in newtype\text{-}of(st\text{-}env(tyn)) \\ \text{(b)} & int \in range\text{-}set[\![tag/(lb\underline{\ldots}ub)]\!]\end{array}}{st\text{-}env \vdash_{Enumerated} [\![tag/int]\!] \text{ has type } tyn}
$$

$$
\frac{\begin{array}{ll}\text{(a)} & tyn \in \mathsf{dom}\, st\text{-}env \wedge \\ & \quad [\![tag/(ch_1,\ldots,ch_{k+1})]\!] \in newtype\text{-}of(st\text{-}env(tyn)) \\ \text{(b)} & char \in range\text{-}set[\![(ch_1,\ldots,ch_{k+1})]\!] \end{array}}{st\text{-}env \vdash_{Enumerated} [\![tag\ '\ char]\!] \text{ has type } tyn} \quad \boxed{\text{wf-Enumerated: EnumChar}}
$$

$$
\frac{\begin{array}{ll}\text{(a)} & tyn \in \mathsf{dom}\, st\text{-}env \wedge \\ & \quad [\![\texttt{STRING}\ [n] ch\text{-}tyn]\!] \in newtype\text{-}of(st\text{-}env(tyn)) \\ \text{(b)} & chr\text{-}nty \in newtype\text{-}of(st\text{-}env(ch\text{-}tyn)) \\ \text{(c)} & tag \in tags\text{-}of\text{-}newtype(chr\text{-}nty) \\ \text{(d)} & \forall\ i \in \{1,\ldots,k+1\} \bullet ch_i \in range\text{-}set[\![chr\text{-}nty]\!] \\ \text{(e)} & k+1 = n \end{array}}{st\text{-}env \vdash_{Enumerated} [\![tag''\, ch_1\ \ldots\ ch_{k+1}\ '']\!] \text{ has type } tyn} \quad \boxed{\text{wf-Enumerated: StringEnum}}
$$

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 uses the operation $make\text{-}constant_{[TY]}(enum)$ to return a basic single-state, single-transition automaton that has no inputs, and has the value $enum \in TY$ on its output:

Dynamic semantics equations:

$M\text{-}Enumerated[\![tag]\!](env) = make\text{-}constant_{[TY]}[\![tag]\!]$

where

$tag \in extract\text{-}set(TY)$

$TY \in Tag^*$

$TY \in \mathsf{rng}\, env$

$M\text{-}Enumerated[\![tag/int]\!](env) =$

$make\text{-}constant_{[TY]}(label\text{-}value(tag, int))$

where

$label\text{-}value(tag, int) \in TY$

$TY \in \mathsf{rng}\, env$

$M\text{-}Enumerated[\![tag\ '\ chr]\!](env) =$

$make\text{-}constant_{[TY]}(label\text{-}value(tag, chr))$

where

$label\text{-}value(tag, chr) \in TY$

$TY \in \mathsf{rng}\, env$

$M\text{-}Enumerated[\![tag\ ''\ ch_1\ \ldots\ ch_{k+1}''\ ]\!](env) =$

$make\text{-}constant_{[TY]}(label\text{-}value(tag, \langle ch_1, \ldots, ch_{k+1}\rangle))$

where

$label\text{-}value(tag, \langle ch_1, \ldots, ch_{k+1}\rangle)) \in TY$

$TY \in \mathsf{rng}\, env$

The following primitive $make\text{-}constant_{[TY]}$ used above is defined below:

$make\text{-}constant_{[TY]} : TY \rightarrow GEN\text{-}IOA$

$make\text{-}constant_{[TY]}(v) = A$

where

$\Sigma_I(A) = \{\,\}$

$\Sigma_O(A) = \{0 \mapsto TY\}$

$States(A) = \{0\}$

$Initials(A) = \{0\}$

$Transitions(A) = \{(0 \xrightarrow{\{0 \mapsto v\}} 0)\}$

pre

$v \in TY$

5.5 Primitives

Structure:

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

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

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

$$\mathit{M\text{-}Primitive} : \mathit{Primitive} \times \mathit{Environment} \rightarrow \mathit{GEN\text{-}IOA}$$

5.5.1 Ambiguity Delay

Structure:

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

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:

$$\frac{\begin{array}{ll} \text{(a)} & del_1 \geq 1 \wedge del_2 \geq 1 \wedge del_1 \leq del_2 \\ \text{(b)} & \textit{st-env} \vdash_{Unit} unm_1 \text{ has type } ty \\ \text{(c)} & \textit{st-env} \vdash_{Unit} unm_2 \text{ has type } ty \\ \text{(d)} & \textit{st-env} \vdash_{\textit{is-const-value}} unm_1 \\ \text{(e)} & \textit{st-env} \vdash_{\textit{is-const-value}} unm_2 \end{array}}{\textit{st-env} \vdash_{Primitive} [\![\, \texttt{DELAY}\,(unm_1, del_1, unm_2, del_2)]\!] \text{ has type } ty} \;\boxed{\text{wf-Primitive: Adelay}}$$

The ambiguity delay is well-formed precisely when:

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

Dynamic semantics equations:

$$\begin{array}{l} \textit{M-Primitive}[\![\, \texttt{DELAY}\,(unm_1, del_1, unm_2, del_2)]\!](env) = \\ \quad \textit{make-adelay}_{[TY]}(\textit{M-Unit}[\![unm_1]\!](env), del_1, \textit{M-Unit}[\![unm_2]\!](env), del_2) \end{array}$$

The operation $\textit{make-adelay}_{[TY]}$ creates a basic automaton which represents the semantics of the ambiguity delay function:

$make\text{-}adelay_{[TY]} : GEN\text{-}IOA \times \mathbb{N} \times GEN\text{-}IOA \times \mathbb{N} \rightarrow GEN\text{-}IOA$

$make\text{-}adelay_{[TY]}(c_1, n_1, c_2, n_2) = A$

where

$k = n_1 + n_2 - 1$

$\Sigma_I(A) = \{1 \mapsto TY\}$

$\Sigma_O(A) = \{0 \mapsto TY\}$

$States(A) = \{lst \mid lst \in TY^* \wedge \mathsf{len}\, lst = k\}$

$Initials(A) = \{[x_1, \ldots, x_k] \mid (\forall\, i \in \{1, \ldots, k\} \bullet x_i = a) \wedge a \in out\text{-}values(c_1)\}$

$\forall\, a_1, \ldots, a_k, v \in TY \bullet$

(a) $subsequence_{[TY]}(n_2, [a_1, \ldots, a_k])$

(b) $m = \{1 \mapsto v, 0 \mapsto a_{n_2}\}$

Adelay-1

$$[a_1, \ldots, a_k] \xrightarrow{m} [v, a_1, \ldots, a_{k\text{-}1}] \in Transitions(A)$$

$\forall\, a_1, \ldots, a_k, v_1 \in TY, v_2 \in out\text{-}values(c_2) \bullet$

(a) $\neg\, subsequence_{[TY]}(n_2, [a_1, \ldots, a_k])$

(b) $m = \{1 \mapsto v_1, 0 \mapsto v_2\}$

Adelay-2

$$[a_1, \ldots, a_k] \xrightarrow{m} [v_1, a_1, \ldots, a_{k\text{-}1}] \in Transitions(A)$$

where $subsequence(k, l)$ is true if there is a same-value subsequence with at least k elements in l, and false otherwise:

$subsequence_{[TY]} : \mathbb{N} \times TY^* \rightarrow \mathbb{B}$

$subsequence_{[TY]}(k, lst) = \exists s_1, s_2 \in TY^*, v \in TY \cdot\ lst = s_1 \frown [v]^k \frown s_2$

Each state *lst* stores the last $n_1 + n_2 - 1$ inputs received. If there has been a sequence of at least n_2 same-value inputs a within this store, then the value a is output (Rule *Adelay*-1), otherwise a value v_2 is output (Rule *Adelay*-2), where $v_2 \in out\text{-}values(c_2)$ — this may be a set of values. The set of initial states is determined by loading the store with $n_1 + n_2 - 1$ copies of $v_1 \in out\text{-}values(c_1)$, which may be a set of values.

The operation $out\text{-}values(c)$ is used to find the set of possible output values from the automaton c — these form the set of initial values for the delay; this will either be a singleton if *unm* returns an enumerated value, or the complete set of values *TY* if *unm* is 'query'. For the latter, then *every* state is in the set of initial states.

5.5.2 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:

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

The inertial delay is well-formed precisely when:

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

Dynamic semantics equations:

$$\textit{M-Primitive}[\![\ \texttt{IDELAY}\ (\textit{unm}, \textit{del})]\!](\textit{env}) = \textit{make-idelay}_{[TY]}(\textit{MU}, \textit{del})$$

where

$$\textit{MU} = \textit{M-Unit}[\![\textit{unm}]\!](\textit{env})$$

$$\textit{TY} = \textit{out-type}(\textit{MU})$$

The operation $\textit{make-idelay}_{[TY]}$ creates a basic automaton which represents the semantics of the inertial delay function for type *TY*:

$make\text{-}idelay_{[TY]} : GEN\text{-}IOA \times \mathbb{N} \to GEN\text{-}IOA$

$make\text{-}idelay_{[TY]}(c,n) = A$

where

$\Sigma_I(A) = \{1 \mapsto TY\}$

$\Sigma_O(A) = \{0 \mapsto TY\}$

$States(A) = \{\langle z,a,b\rangle \mid z \in \{1,\ldots,n\} \wedge a,b \in TY\}$

$Initials(A) = \{\langle n,a,a\rangle \mid a \in out\text{-}values(c)\}$

$\forall a,b,v{:}\,TY, i{:}\,\{1,\ldots,n\}\cdot$

$$\boxed{\text{Idelay-1}}\ \frac{v = a \wedge i = n}{\langle i,a,b\rangle \xrightarrow{\{1\mapsto v, 0\mapsto a\}} \langle i,a,a\rangle \in Transitions(A)}$$

$$\boxed{\text{Idelay-2}}\ \frac{v = a \wedge i \neq n}{\langle i,a,b\rangle \xrightarrow{\{1\mapsto v, 0\mapsto b\}} \langle i+1,a,b\rangle \in Transitions(A)}$$

$$\boxed{\text{Idelay-3}}\ \frac{v \neq a \wedge i = n}{\langle i,a,b\rangle \xrightarrow{\{1\mapsto v, 0\mapsto a\}} \langle 1,v,a\rangle \in Transitions(A)}$$

$$\boxed{\text{Idelay-4}}\ \frac{v \neq a \wedge i \neq n}{\langle i,a,b\rangle \xrightarrow{\{1\mapsto v, 0\mapsto b\}} \langle 1,v,b\rangle \in Transitions(A)}$$

Each state $\langle i,a,b\rangle$ has a counter i which indicates the number of same-value inputs received. Store a holds the last input value, and store b holds the last output value. Rule *Idelay*-1 is followed when at least n same-value inputs a have been received and the output value is held at, or changes to, a. Rule *Idelay*-2 is followed when the input value is the same as the previous value a, but fewer than n values have been received; here the counter is incremented. Rule *Idelay*-3 is followed when n same-value inputs a have been received and the output value changes to a. However, then new input is different from the previous one, so the counter i is reset to 1. Rule *Idelay*-4 is followed when fewer than n same-value inputs a have been received, but the new input is different. Here the counter is reset to 1 and the previous input(s) a are ignored.

An initial state is determined by setting the state as if at least n inputs of value v had been received during *imaginary* time-units $-n,\ldots,-1$, where $v \in out\text{-}values(c)$. The values of a and b will therefore both be set to v and the counter will be set to n.

5.5.3 Sample

Structure:

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

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:

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

The sample is well-formed precisely when:

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

Dynamic semantics equations:

$$\textit{M-Primitive}[\![\ \texttt{SAMPLE}\,(n_1, unm, i_2)]\!](env) = \textit{make-sample}_{[TY]}(n_1, MU, i_2)$$

where

$$MU = \textit{M-Unit}[\![unm]\!](env)$$

$$TY = \textit{out-type}(MU)$$

The operation $\textit{make-sample}_{[TY]}$ creates a basic automaton which represents the semantics of the sampling function:

$make\text{-}sample_{[TY]} : \mathbb{N} \times GEN\text{-}IOA \times \mathbb{Z} \rightarrow GEN\text{-}IOA$

$make\text{-}sample_{[TY]}(n_1, c, i_2) = A$

where

$\Sigma_I(A) = \{1 \mapsto TY\}$

$\Sigma_O(A) = \{0 \mapsto TY\}$

$States(A) = \{\langle i, a\rangle \mid i \in \{0, \ldots, n_1 - 1\} \wedge a \in TY\}$

$Initials(A) = \{\langle z, x\rangle \mid z = (n_1 - i_2) \bmod n_1 \wedge x \in out\text{-}values(c)\}$

$\forall a, v{:}\, ty, i{:}\, \{0, \ldots, n_1 - 1\} \cdot$

$$\boxed{\text{Sample-1}}\ \frac{i = 0}{(\langle i, a\rangle \xrightarrow{\{1 \mapsto v, 0 \mapsto v\}} \langle 1, v\rangle) \in Transitions(A)}$$

$$\boxed{\text{Sample-2}}\ \frac{i \neq 0}{(\langle i, a\rangle \xrightarrow{\{1 \mapsto v, 0 \mapsto a\}} \langle (i+1) \bmod n_1, a\rangle) \in Transitions(A)}$$

Each state $\langle i, a\rangle$ has a counter i which indicates the number of inputs since the last sample point — when $i = 0$ then a new input sample is taken, and stored as part of the state in a. Rule *Sample*-1 gives the transition relations when a sample is taken, and rule *Sample*-2 gives the non-sampling transitions. Any skew is achieved by setting a to a initial constant value $x \in out\text{-}values(c)$ and setting the *counter* to the required position in a sampling cycle.

5.6 Unit

Structure:

$$\begin{array}{|lcl|} \hline \textit{Unit} & ::= & \textit{enumerated} \mid \textit{signal} \mid \textit{assoc} \mid \textit{unitstr} \mid \textit{replicate} \\ & \mid & \textit{query} \mid \textit{index} \mid \textit{dyindex} \mid \textit{replace} \mid \textit{trim} \\ & \mid & \textit{conc} \mid \textit{sequence} \mid \textit{extract} \mid \textit{instance} \mid \textit{let} \mid \textit{case} \\ & \mid & \textit{unitvoid} \\ \hline \end{array}$$

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

$$(\_ \vdash_{Unit} \_ \text{ has type } \_) \subseteq (\textit{StaticEnvironment} \times \textit{Unit} \times \textit{Type})$$
$$\textit{M-Unit} : \textit{Unit} \times \textit{Environment} \rightarrow \textit{GEN-IOA}$$

A dereferencing rule is required so that a type can be related to any associated typename during static checking of units:

Static semantics rules:

$$\boxed{\text{wf-Unit : type deref}} \quad \dfrac{\begin{array}{ll} \text{(a)} & \textit{tynm} \in \text{dom}\, \textit{st-env} \\ \text{(b)} & \textit{st-env}(\textit{tynm}) = [\![\textit{tynm} = \textit{ty}]\!] \\ \text{(c)} & \textit{st-env} \vdash_{Unit} \textit{un} \text{ has type } \textit{ty} \end{array}}{\textit{st-env} \vdash_{Unit} \textit{un} \text{ has type } \textit{tynm}}$$

5.6.1 UnitVoid

The semantics for unitvoid is represented by a basic automaton producing the constant `unitvoid` $\in$ *VOIDTYPE*:

Static semantics rules:

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

Dynamic semantics equations:

$$\textit{M-Unit}[\![(\ )]\!](\textit{env}) = \textit{make-constant}_{[\textit{TyVoid}]}(\ \texttt{unitvoid}\ )$$

where

$$\textit{TyVoid} = \textit{M-Type}[\![(\ )]\!](\textit{env})$$

5.6.2 Enumerated Unit

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

Static semantics rules:

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

Dynamic semantics equations:

$$\textit{M-Unit}[\![\textit{en}]\!](\textit{env}) = \textit{M-Enumerated}[\![\textit{en}]\!](\textit{env})$$

5.6.3 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:

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

Assoc uses the operation $assoc_{[ITY,OTY]}(tag)$ to build an automaton which tags each input value $v \in ITY$ with a tag $tag \in Tag$:

Dynamic semantics equations:

$$M\text{-}Unit[\![en \;\&\; unm]\!](env) =$$
$$(MU[0 \mapsto 1] \,||\, assoc_{[ITY,OTY]}(en)) \backslash \{1\}$$

where

$$MU = M\text{-}Unit[\![unm]\!](env)$$
$$ITY = out\text{-}type(MU)$$
$$label\text{-}type(en, ITY) \subset OTY$$
$$OTY \in GEN\text{-}TYPE$$

$$assoc_{[ITY,OTY]} : Tag \rightarrow GEN\text{-}IOA$$
$$assoc_{[ITY,OTY]}(tag) = A$$

where

$$\Sigma_I(A) = \{1 \mapsto ITY\}$$
$$\Sigma_O(A) = \{0 \mapsto OTY\}$$
$$States(A) = \{0\}$$
$$Initials(A) = \{0\}$$
$$\forall v\!: ITY \cdot$$

$$\boxed{\text{assoc}}\ \frac{w = label\text{-}value(tag, v)}{0 \xrightarrow{\{1 \mapsto v,\, 0 \mapsto w\}} 0 \in Transitions(A)}$$

5.6.4 UnitStr

Structure:

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

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:

$$\frac{\begin{array}{ll}\text{(a)} & n \in \mathbb{N} \wedge n \geq 1 \\ \text{(b)} & \mathit{st\text{-}env} \vdash_{\mathit{Unit}} \mathit{unm} \text{ has type } \mathit{cty} \\ \text{(c)} & \mathit{st\text{-}env} \vdash_{\mathit{is\text{-}char\text{-}type}} \mathit{cty} \\ \text{(d)} & \mathit{tynm} \in \mathrm{dom}\, \mathit{st\text{-}env} \wedge \mathit{st\text{-}env} \vdash_{\mathit{Type\text{-}Equality}} \mathit{tynm} \equiv [\![\, \texttt{STRING}\ [n]\ \mathit{cty}]\!]\end{array}}{\mathit{st\text{-}env} \vdash_{\mathit{Unit}} [\![\, \texttt{STRING}\ [n]\mathit{unm}]\!] \text{ has type } \mathit{tynm}}\ \boxed{\text{wf-Unit: Unitstr}}$$

Unitstr is well-formed when:

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

Dynamic semantics equations:

$$\begin{array}{l} \mathit{M\text{-}Unit}[\![\, \texttt{STRING}\ [n]\mathit{unm}]\!](\mathit{env}) = \\ \quad (\mathit{MU}[0 \mapsto 1] \,||\, \mathit{unit\text{-}string}_{[\mathit{ITY},\mathit{OTY}]}(n)) \backslash \{1\} \\ \quad\quad \text{where} \\ \quad\quad\quad \mathit{MU} = \mathit{M\text{-}Unit}[\![\mathit{unm}]\!](\mathit{env}) \\ \quad\quad\quad \mathit{ITY} = \mathit{out\text{-}type}(\mathit{MU}) \\ \quad\quad\quad \mathit{OTY} = \mathit{M\text{-}Type}[\![\, \texttt{STRING}\ [n]\ \mathit{ITY}]\!](\mathit{env}) \end{array}$$

Unitstr uses the operation $\mathit{unit\text{-}string}_{[\mathit{ITY},\mathit{OTY}]}(n)$ to build an automaton which produces an n-bit string of type *OTY*, containing characters from type *ITY*:

$\mathit{unit\text{-}string}_{[\mathit{ITY},\mathit{OTY}]} : \mathbb{N} \to \mathit{GEN\text{-}IOA}$

$\mathit{unit\text{-}string}_{[\mathit{ITY},\mathit{OTY}]}(n) = A$

where

$\Sigma_I(A) = \{1 \mapsto \mathit{ITY}\}$

$\Sigma_O(A) = \{0 \mapsto \mathit{OTY}\}$

$\mathit{States}(A) = \{0\}$

$\mathit{Initials}(A) = \{0\}$

$\forall v : \mathit{ITY} \cdot$

$$\boxed{\text{unit-string}}\quad \frac{\begin{array}{ll}\text{(a)} & \mathit{label\text{-}value}(\mathit{tag}, w) = v \\ \text{(b)} & \forall i \in \{1, \ldots, n\} \cdot a_i = w \wedge z = \mathit{label\text{-}value}(\mathit{tag}, \langle a_1, \ldots, a_n \rangle)\end{array}}{0 \xrightarrow{\{1 \mapsto v, 0 \mapsto z\}} 0 \in \mathit{Transitions}(A)}$$

5.6.5 Replicate

Structure:

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

The replication operation $[\ k\ ]\ \textit{unm}$ returns a tuple value where each component contains the value produced by *unm*. The type of the resulting unit will be the product of the type of *unm*.

Static semantics rules:

$$\boxed{\text{wf-Unit: Replicate}}\ \frac{\begin{array}{ll} \text{(a)} & k \in \mathbb{N} \wedge k \geq 1 \\ \text{(b)} & \textit{st-env} \vdash_{\textit{Unit}}\ \textit{unm}\ \text{has type}\ \textit{ty} \\ \text{(c)} & \textit{row-ty} = [\![(\textit{ty}_1, \ldots, \textit{ty}_k)]\!] \\ \text{(d)} & \forall\ i \in \{1, \ldots, k\} \bullet \textit{ty}_i = \textit{ty} \end{array}}{\textit{st-env} \vdash_{\textit{Unit}} [\![[\ k\ ]\ \textit{unm}]\!]\ \text{has type}\ \textit{row-ty}}$$

Dynamic semantics equations:

$$\begin{array}{l} \textit{M-Unit}[\![[\ k\ ]\ \textit{unm}]\!](\textit{env}) = \\ \quad (\textit{MU}_1[0 \mapsto 1] \,||\, \ldots \,||\, \textit{MU}_k[0 \mapsto k] \,||\, \textit{make-sequence}_{[\textit{OTY}]}) \backslash \{1, \ldots, k\} \\ \qquad \text{where} \\ \qquad\quad \textit{MU} = \textit{M-Unit}[\![\textit{unm}]\!](\textit{env}) \\ \qquad\quad \textit{TY} = \textit{out-type}(\textit{MU}) \\ \qquad\quad \textit{OTY} = [\textit{ty}_1, \ldots, \textit{ty}_k] \\ \qquad\quad \forall\ i \in \{1, \ldots, k\} \bullet \textit{ty}_i = \textit{TY} \\ \qquad\quad \forall\ i \in \{1, \ldots, k\} \bullet \textit{MU}_i = \textit{MU} \end{array}$$

5.6.6 Query

Structure:

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

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:

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

Query uses the operation $\mathit{make\text{-}query}_{[\mathit{TY}]}$ to build a one-state automaton which has an empty input signature and $\mathsf{card}\,\mathit{TY}$ transitions, each one having as an output a different value from *TY*:

Dynamic semantics equations:

$$\mathit{M\text{-}Unit}[\![\ ?\ \mathit{ty}]\!](\mathit{env}) = \mathit{make\text{-}query}_{[\mathit{TY}]}$$

where

$$\mathit{TY} = \mathit{M\text{-}Type}[\![\mathit{ty}]\!](\mathit{env})$$

$$\mathit{make\text{-}query}_{[\mathit{TY}]} : \mathit{GEN\text{-}IOA}$$

$$\mathit{make\text{-}query}_{[\mathit{TY}]} = A$$

where

$$\Sigma_I(A) = \{\,\}$$

$$\Sigma_O(A) = \{0 \mapsto \mathit{TY}\}$$

$$\mathit{States}(A) = \{0\}$$

$$\mathit{Initials}(A) = \{0\}$$

$$\mathit{Transitions}(A) = \{(0 \xrightarrow{\{0 \mapsto a\}} 0) \in \mathit{TRANS} \mid a \in \mathit{TY}\}$$

5.6.7 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 character of type ty, selected by i. The indexer must fall in the range of valid indices for the output of unm.

Static semantics rules:

$$\boxed{\text{wf-Unit: Index-row}}\;\frac{\begin{array}{ll}\text{(a)} & st\text{-}env \vdash_{Unit}\ unm \text{ has type } [\![(ty_1,\ldots,ty_k)]\!] \\ \text{(b)} & 1 \le i \le k\end{array}}{st\text{-}env \vdash_{Unit} [\![unm\ [\ i\ ]]\!](env) \text{ has type } ty_i}$$

$$\boxed{\text{wf-Unit: Index-str}}\;\frac{\begin{array}{ll}\text{(a)} & st\text{-}env \vdash_{Unit}\ unm \text{ has type } [\![\ \texttt{STRING}\ [k]ty]\!] \\ \text{(b)} & 1 \le i \le k\end{array}}{st\text{-}env \vdash_{Unit} [\![unm\ [\ i\ ]]\!](env) \text{ has type } ty}$$

The index operation is well-formed when:

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

The indexing operation uses the function *select-ind* to build an automaton which extracts the required element from the output of unm:

Dynamic semantics equations:

$$M\text{-}Unit[\![unm[\ i\ ]]\!](env) = (MU[0 \mapsto 1]\ ||\ select\text{-}ind_{[ITY,OTY]}(i))\backslash\{1\}$$

where

$$MU = M\text{-}Unit[\![unm]\!](env)$$
$$ITY = out\text{-}type(MU)$$
$$OTY = index\text{-}type(ITY, i)$$

The operation $select\text{-}ind_{[ITY,OTY]}(n)$ builds an automaton which has input signature ITY and output signature OTY. It takes either an input sequence $v = \langle a_1,\ldots,a_k\rangle \in ITY$, and outputs the element a_n, or it takes an input string $v = label\text{-}value(tag, tuplec_1,\ldots,c_k)$ and outputs a character $label\text{-}value(tag, c_n)$.

Two useful types are first given. The type *TAG-STRING* contains sets of ELLA-strings — these are tagged rows, where the elements in the row are the same character-set:

$$TAG\text{-}STRING =$$
$$\{label\text{-}type(tag, chr\text{-}tuple) \in TAG\text{-}TYPE \mid chr\text{-}tuple = tuple\text{-}product(\langle ty_1, \ldots, ty_k \rangle) \wedge$$
$$(\forall\, i \in \{1, \ldots, k\} \bullet ty_i = typ) \wedge typ \in CHAR\text{-}SET\}$$

The type *ROW-OR-STRING* combines rows and ELLA-strings:

$$ROW\text{-}OR\text{-}STRING = ROW \cup TAG\text{-}STRING$$

$select\text{-}ind_{[ITY,OTY]} : \mathbb{N} \to GEN\text{-}IOA$

$select\text{-}ind_{[ITY,OTY]}(n) = A$

where

$\Sigma_I(A) = \{1 \mapsto ITY\}$

$\Sigma_O(A) = \{0 \mapsto OTY\}$

$States(A) = \{0\}$

$Initials(A) = \{0\}$

$\forall\, v : ITY \bullet$

$$\boxed{\text{select-ind: row}}\ \frac{\text{(a)}\quad \langle a_1, \ldots, a_k \rangle = v \qquad \text{(b)}\quad m = \{1 \mapsto v, 0 \mapsto a_n\}}{(0 \xrightarrow{m} 0) \in Transitions(A)}$$

$$\boxed{\text{select-ind: str}}\ \frac{\text{(a)}\quad label\text{-}value(tag, \langle a_1, \ldots, a_k \rangle) = v \qquad \text{(b)}\quad w = label\text{-}value(tag, a_n) \qquad \text{(c)}\quad m = \{1 \mapsto v, 0 \mapsto w\}}{(0 \xrightarrow{m} 0) \in Transitions(A)}$$

pre

$ITY \in ROW\text{-}OR\text{-}STRING \wedge$

$k = row\text{-}string\text{-}len(ITY) \wedge$

$OTY = index\text{-}type(ITY, n) \wedge$

$n \in \{1, \ldots, k\}$

The operation $index\text{-}type(ty, n)$ accepts a row- or string-type ty and returns the type element indexed by n:

$$index\text{-}type : (ROW\text{-}OR\text{-}STRING) \times \mathbb{N} \rightarrow GEN\text{-}TYPE$$

$$index\text{-}type(tuple\text{-}product(\langle ty_1, \ldots, ty_k \rangle)), n) = ty_n$$

$$index\text{-}type(label\text{-}type(tag, tuple\text{-}product(\langle ty_1, \ldots, ty_k \rangle))), n) =$$

$$label\text{-}type(tag, ty)$$

where

$$\forall\, i \in \{1, \ldots, k\} \bullet ty_i = ty$$

$$ty \in Char$$

pre

$$n \in \{1, \ldots, k\}$$

The operation $row\text{-}string\text{-}len(ty)$ accepts a row- or string-type ty and returns the length of the row/string elements contained in that type:

$$row\text{-}string\text{-}len : (ROW\text{-}OR\text{-}STRING) \rightarrow \mathbb{N}$$

$$row\text{-}string\text{-}len(tuple\text{-}product(\langle ty_1, \ldots, ty_k \rangle)) = k$$

$$row\text{-}string\text{-}len(label\text{-}type(tag, tuple\text{-}product(\langle ty_1, \ldots, ty_k \rangle))) = k$$

5.6.8 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:

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

$$\dfrac{\begin{array}{ll} \text{(a)} & \mathit{st\text{-}env} \vdash_{\mathit{Unit}} \mathit{unm}_1 \text{ has type } [\![\; \texttt{STRING}\; [k]\mathit{ty}]\!] \\ \text{(b)} & \mathit{st\text{-}env} \vdash_{\mathit{Unit}} \mathit{unm}_2 \text{ has type } \mathit{nty} \\ \text{(c)} & \mathit{st\text{-}env} \vdash_{\mathit{Type\text{-}Equality}} \mathit{nty} \equiv [\![\mathit{tag}/(\mathit{lb}\,\underline{..}\,\mathit{ub})]\!] \\ \text{(d)} & 1 \leq \mathit{lb} \wedge \mathit{lb} \leq \mathit{ub} \wedge \mathit{ub} \leq k \end{array}}{\mathit{st\text{-}env} \vdash_{\mathit{Unit}} [\![\mathit{unm}_1 \; [[\; \mathit{unm}_2 \;]]]\!] \text{ has type } \mathit{ty}} \quad \boxed{\text{wf-Unit: Dyindex-str}}$$

The dynamic index operation is well-formed when:

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

The dynamic index uses the operation *select-dy* to build an automaton which extracts the required element:

Dynamic semantics equations:

$M\text{-}Unit[\![unm_1\ [[\ unm_2\ ]]]\!](env) =$

$(MU1[0 \mapsto 1] \,||\, MU2[0 \mapsto 2] \,||\, select\text{-}dy_{[ITY,NTY,OTY]})\backslash\{1,2\}$

where

$MU1 = M\text{-}Unit[\![unm_1]\!](env)$

$MU2 = M\text{-}Unit[\![unm_2]\!](env)$

$ITY = out\text{-}type(MU1)$

$NTY = out\text{-}type(MU2)$

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

The operation $select\text{-}dy_{[ITY,NTY,OTY]}$ builds an automaton which has input signature $ITY \times NTY$ and output signature OTY. It takes as input a sequence or string v, and an indexer $ind = label\text{-}value(tag_1, i)$. If v is a sequence $\langle a_1, \ldots, a_k \rangle \in ITY$, then the element a_i is produced; if v is a string $label\text{-}value(tag_2, tuplec_1, \ldots, c_k)$ then it a character $label\text{-}value(tag_2, c_i)$ is produced:

$select\text{-}dy_{[ITY,NTY,OTY]} : GEN\text{-}IOA$

$select\text{-}dy_{[ITY,NTY,OTY]} = A$

where

$\Sigma_I(A) = \{1 \mapsto ITY, 2 \mapsto NTY\}$

$\Sigma_O(A) = \{0 \mapsto OTY\}$

$States(A) = \{0\}$

$Initials(A) = \{0\}$

$\forall\ v{:}ITY, ind \in NTY \bullet$

$$\boxed{\text{select-dy: row}}\ \frac{\begin{array}{ll}\text{(a)} & label\text{-}value(tag, i) = ind \\ \text{(b)} & \langle a_1, \ldots, a_k \rangle = v \\ \text{(c)} & m = \{1 \mapsto v, 2 \mapsto ind, 0 \mapsto a_i\}\end{array}}{(0 \xrightarrow{m} 0) \in Transitions(A)}$$

$$\boxed{\text{select-dy: str}}\ \frac{\begin{array}{ll}\text{(a)} & label\text{-}value(tag_1, i) = ind \\ \text{(b)} & label\text{-}value(tag_2, \langle a_1, \ldots, a_k \rangle) = v \\ \text{(c)} & w = label\text{-}value(tag_2, a_i) \\ \text{(d)} & m = \{1 \mapsto v, 2 \mapsto ind, 0 \mapsto w\}\end{array}}{(0 \xrightarrow{m} 0) \in Transitions(A)}$$

pre

$ITY \in ROW\text{-}OR\text{-}STRING\ \wedge$

$k = row\text{-}string\text{-}len(ITY)\ \wedge$

$(\forall\ i \in \{1, \ldots, k\} \bullet OTY = index\text{-}type(ITY, i))\ \wedge$

$NTY \in TAG\text{-}INT\text{-}RANGE \wedge$

$min\text{-}int\text{-}rng(NTY) \geq 1 \wedge max\text{-}int\text{-}rng(NTY) \leq k$

5.6.9 Replace

Structure:

$$\boxed{\textit{Replace} \quad ::= \quad \texttt{REPLACE}\,(unit_1, unit_2, unit_3)}$$

The replacement operation $\texttt{REPLACE}\,(unm_1, unm_2, unm_3)$ 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 also takes a replacement unit unm_3, which has the same output type ty, 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:

$$\frac{\begin{array}{ll} \text{(a)} & \textit{st-env} \vdash_{Unit}\ unm_1 \text{ has type } [\![[k]ty]\!] \\ \text{(b)} & \textit{st-env} \vdash_{Unit}\ unm_2 \text{ has type } nty \\ \text{(c)} & \textit{st-env} \vdash_{Unit}\ unm_3 \text{ has type } ty \\ \text{(d)} & \textit{st-env} \vdash_{Type\text{-}Equality}\ nty \equiv [\![tag/(lb\_\_ub)]\!] \\ \text{(e)} & 1 \leq lb \wedge lb \leq ub \wedge ub \leq k \end{array}}{\textit{st-env} \vdash_{Unit} [\![\ \texttt{REPLACE}\,(unm_1, unm_2, unm_3)]\!] \text{ has type } [\![[k]ty]\!]}\ \boxed{\text{wf-Unit: Replace-row}}$$

$$\frac{\begin{array}{ll} \text{(a)} & \textit{st-env} \vdash_{Unit}\ unm_1 \text{ has type } [\![\ \texttt{STRING}\ [k]ty]\!] \\ \text{(b)} & \textit{st-env} \vdash_{Unit}\ unm_2 \text{ has type } nty \\ \text{(c)} & \textit{st-env} \vdash_{Unit}\ unm_3 \text{ has type } ty \\ \text{(d)} & \textit{st-env} \vdash_{Type\text{-}Equality}\ nty \equiv [\![tag/(lb\_\_ub)]\!] \\ \text{(e)} & 1 \leq lb \wedge lb \leq ub \wedge ub \leq k \end{array}}{\textit{st-env} \vdash_{Unit} [\![\ \texttt{REPLACE}\,(unm_1, unm_2, unm_3)]\!] \text{ has type } [\![\ \texttt{STRING}\ [k]ty]\!]}\ \boxed{\text{wf-Unit: Replace-str}}$$

The replacement operation is well-formed when:

1. unm_1 is either a sequence where each element is of type ty, or unm_1 outputs a string,
2. unm_2 returns an ELLA-integer which falls in the range of valid indices,
3. the output of unm_3 has type ty.

Replace uses the operation $replace_{[STY,NTY,RTY]}$ to build an automaton which inserts the new output unit at the required index:

Dynamic semantics equations:

$$M\text{-}Unit[\![\ \texttt{REPLACE}\ (unm_1, unm_2, unm_3)]\!](env) =$$

$$(MU_1[0 \mapsto 1] \mid\mid MU_2[0 \mapsto 2] \mid\mid MU_3[0 \mapsto 3] \mid\mid replace_{[STY,NTY,RTY]}) \backslash \{1,2,3\}$$

where

$$MU_1 = M\text{-}Unit[\![unm_1]\!](env)$$
$$MU_2 = M\text{-}Unit[\![unm_2]\!](env)$$
$$MU_3 = M\text{-}Unit[\![unm_3]\!](env)$$
$$STY = out\text{-}type(MU1)$$
$$tuple\text{-}product(\langle ty_1, \ldots, ty_k \rangle) = STY$$
$$NTY = out\text{-}type(MU2)$$
$$\forall\ i \in \{1, \ldots, k\} \bullet ty_i = RTY$$

The operation $replace_{[STY,NTY,RTY]}$ builds an automaton which has input signature $STY \times NTY \times RTY$ and output signature STY. It takes as input a sequence or string $v \in STY$, an indexer $ind = label\text{-}value(tag_1, i)$, and a replacement value $x \in RTY$. If v is a sequence $\langle a_1, \ldots, a_k \rangle$, then the seqeuence $w = \langle a_1, \ldots, a_{i\text{-}1}, x, a_{i+1}, \ldots, a_k \rangle$ is produced; if v is a string $label\text{-}value(tag_2, \langle c_1, \ldots, c_k \rangle)$ then the string $label\text{-}value(tag_2, \langle c_1, \ldots, c_{i\text{-}1}, x, c_{i+1}, \ldots, c_k \rangle)$ is produced:

$replace_{[STY,NTY,RTY]}$: *GEN-IOA*

$replace_{[STY,NTY,RTY]} = A$

where

$\Sigma_I(A) = \{1 \mapsto STY, 2 \mapsto NTY, 3 \mapsto RTY\}$

$\Sigma_O(A) = \{0 \mapsto STY\}$

$States(A) = \{0\}$

$Initials(A) = \{0\}$

$\forall\ v : STY, ind : NTY, x : RTY \bullet$

(a) $\langle a_1, \ldots, a_k \rangle = v$

(b) $label\text{-}value(tag, i) = ind$

(c) $z = \langle a_1, \ldots, a_{i\text{-}1}, x, a_{i+1}, \ldots, a_k \rangle$

(d) $m = \{1 \mapsto v, 2 \mapsto ind, 3 \mapsto x, 0 \mapsto z\}$

replace: row

$(0 \xrightarrow{m} 0) \in Transitions(A)$

(a) $label\text{-}value(tag_1, i) = ind$

(b) $label\text{-}value(tag_2, \langle a_1, \ldots, a_k \rangle) = v$

(c) $label\text{-}value(tag_2, w) = w$

(d) $z = label\text{-}value(tag_2, \langle a_1, \ldots, a_{i\text{-}1}, w, a_{i+1}, \ldots, a_k \rangle)$

(e) $m = \{1 \mapsto v, 2 \mapsto ind, 3 \mapsto 8, 0 \mapsto z\}$

replace: str

$(0 \xrightarrow{m} 0) \in Transitions(A)$

pre

$STY \in ROW\text{-}OR\text{-}STRING \wedge$

$k = row\text{-}string\text{-}len(ITY) \wedge$

$(\forall\ i \in \{1, \ldots, k\} \bullet RTY = index\text{-}type(STY, i)) \wedge$

$NTY \in TAG\text{-}INT\text{-}RANGE \wedge$

$min\text{-}int\text{-}rng(NTY) \geq 1 \wedge max\text{-}int\text{-}rng(NTY) \leq k$

5.6.10 Trim

Structure:

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

The trimming operation $unm[i\underline{..}j]$ accepts the output of unm, which must be either a sequence $\langle v_1,\ldots,v_k\rangle$ of type $(ty_1,\ldots,ty_k)$, or a string $\mathit{label\text{-}value}(tag,\langle v_1,\ldots,v_k\rangle)$ over characters of type ty. For a sequence, it returns the subsequence of type $(ty_i,\ldots,ty_j)$. For the string, it returns the substring $\mathit{label\text{-}value}(tag,\langle v_i,\ldots,v_j\rangle)$. Clearly, $i \leq j$, and the two indices must fall in the range of valid indices from unm:

Static semantics rules:

$$\frac{\begin{array}{ll}(a) & \mathit{st\text{-}env} \vdash_{\mathit{Unit}} \mathit{unm} \text{ has type } [\![ty_1,\ldots,ty_k)]\!] \\ (b) & m = j - i + 1 \\ (c) & \mathit{st\text{-}env} \vdash_{\mathit{Type\text{-}Equality}} ty \equiv [\![(ty_i,\ldots,ty_j)]\!] \\ (d) & j \in \{1,\ldots,k\} \\ (e) & i \in \{1,\ldots,j\}\end{array}}{\mathit{st\text{-}env} \vdash_{\mathit{Unit}} [\![\mathit{unm}[i\underline{..}j\,]]\!] \text{ has type } ty}\ \boxed{\text{wf-Unit: Trim-row}}$$

$$\frac{\begin{array}{ll}(a) & \mathit{st\text{-}env} \vdash_{\mathit{Unit}} \mathit{unm} \text{ has type } [\![\ \texttt{STRING}\ \ [k]typ]\!] \\ (b) & m = j - i + 1 \\ (c) & \mathit{st\text{-}env} \vdash_{\mathit{Type\text{-}Equality}} ty \equiv [\![\ \texttt{STRING}\ \ [m]typ]\!] \\ (d) & j \in \{1,\ldots,k\} \\ (e) & i \in \{1,\ldots,j\}\end{array}}{\mathit{st\text{-}env} \vdash_{\mathit{Unit}} [\![\mathit{unm}[i\underline{..}j\,]]\!] \text{ has type } ty}\ \boxed{\text{wf-Unit: Trim-str}}$$

The trimming operation is well-formed when:

1. Unit *unm* is either a sequence or a string,
2. 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 uses the operation *select-seq* to build an automaton which extracts the required elements or substring from its input:

Dynamic semantics equations:

$$M\text{-}Unit[\![unm[i\_..\_j\,]]\!](env) = (MU[0 \mapsto 1] \,||\, select\text{-}seq_{[ITY,OTY]}(i,j)) \backslash \{1\}$$

where

$MU = M\text{-}Unit[\![unm]\!](env)$

$ITY = out\text{-}type(MU)$

$tuple\text{-}product(\langle ty_1, \ldots, ty_k \rangle) = ITY$

$OTY = trim\text{-}type(ITY, i, j)$

The operation $select\text{-}seq_{[ITY,OTY]}(i,j)$ creates a basic automaton which allows either

- a subsequence of values $v(m, \ldots, n) \in OTY$ to be selected from an input $v \in ITY$, or
- a substring $label\text{-}value(tag, \langle chr_m, \ldots, chr_n \rangle) \in OTY$ to be extracted from an input $label\text{-}value(tag, \langle chr_1, \ldots, chr_k \rangle) \in ITY$.

$select\text{-}seq_{[ITY,OTY]} : \mathbb{N} \times \mathbb{N} \to GEN\text{-}IOA$

$select\text{-}seq_{[ITY,OTY]}(m,n) = A$

where

$\Sigma_I(A) = \{1 \mapsto ITY\}$

$\Sigma_O(A) = \{0 \mapsto OTY\}$

$States(A) = \{0\}$

$Initials(A) = \{0\}$

$\forall v\colon ITY \cdot$

select-seq: Trans-row

$$\frac{\text{(a)}\quad \langle a_1, \ldots, a_k \rangle = v \qquad \text{(b)}\quad w = \langle a_m, \ldots, a_n \rangle}{(0 \xrightarrow{\{1 \mapsto v, 0 \mapsto w\}} 0) \in Transitions(A)}$$

select-seq: Trans-str

$$\frac{\text{(a)}\quad label\text{-}value(tag, \langle c_1, \ldots, c_k \rangle) = v \qquad \text{(b)}\quad w = label\text{-}value(tag, \langle c_m, \ldots, c_n \rangle)}{(0 \xrightarrow{\{1 \mapsto v, 0 \mapsto w\}} 0) \in Transitions(A)}$$

pre

$ITY \in ROW \cup TAG\text{-}TYPE \wedge$

$\quad OTY = trim\text{-}type(ITY, i, j)$

The operation $trim\text{-}type(gty, i, j)$ takes a row- or string-type and returns a new row-/string-type, containing the elements of type $ty_i, \ldots, ty_j$:

$$\mathit{trim\text{-}type} : (\mathit{ROW\text{-}OR\text{-}STRING}) \times \mathbb{N} \times \mathbb{N} \rightarrow (\mathit{ROW\text{-}OR\text{-}STRING})$$

$$\mathit{trim\text{-}type}(\mathit{tuple\text{-}product}(\langle ty_1, \ldots, ty_k \rangle), i, j) =$$
$$\quad \mathit{tuple\text{-}product}(\langle ty_i, \ldots, ty_j \rangle)$$

$$\mathit{trim\text{-}type}(\mathit{label\text{-}type}(\mathit{tag}, \mathit{tuple\text{-}product}(\langle ty_1, \ldots, ty_k \rangle)), i, j) =$$
$$\quad \mathit{label\text{-}type}(\mathit{tag}, \mathit{tuple\text{-}product}(\langle ty_i, \ldots, ty_j \rangle))$$

pre

$$1 \leq i \wedge i \leq j \wedge j \leq k$$

5.6.11 Conc

Structure:

$$\boxed{\mathit{Conc} \quad ::= \qquad \mathit{unit}_1 \ \textsc{conc} \ \mathit{unit}_2}$$

The concatenation operation unm_1 CONC unm_2 takes two units and outputs tuples or strings, which are a concatentation of the outputs from unm_1 and unm_2. Tuples and strings are handled separately — it is not possible to form a *sequence* with a character type, only a string, and vice versa with a non-character type.

$$\boxed{\text{wf-Conc (a)}} \quad \frac{\begin{array}{ll}\text{(a)} & \neg(\mathit{st\text{-}env} \vdash_{\mathit{is\text{-}char\text{-}type}} \mathit{typ}) \\ \text{(b)} & \mathit{st\text{-}env} \vdash_{\mathit{Unit}} \ \mathit{unm} \text{ has type } [\![\mathit{typ}]\!]\end{array}}{\mathit{st\text{-}env} \vdash_{\mathit{Conc}} \mathit{unm} \text{ has type } [\![[1]\mathit{typ}]\!]}$$

$$\boxed{\text{wf-Conc (b)}} \quad \frac{\begin{array}{ll}\text{(a)} & \mathit{st\text{-}env} \vdash_{\mathit{is\text{-}char\text{-}type}} \mathit{typ} \\ \text{(b)} & \mathit{st\text{-}env} \vdash_{\mathit{Unit}} \ \mathit{unm} \text{ has type } [\![\mathit{typ}]\!]\end{array}}{\mathit{st\text{-}env} \vdash_{\mathit{Conc}} \mathit{unm} \text{ has type } [\![\ \textsc{string} \ [1]\mathit{typ}]\!]}$$

$$\boxed{\text{wf-Conc (c)}} \quad \frac{\mathit{st\text{-}env} \vdash_{\mathit{Unit}} \ \mathit{unm} \text{ has type } [\![\mathit{typ}]\!]}{\mathit{st\text{-}env} \vdash_{\mathit{Conc}} \mathit{unm} \text{ has type } [\![\mathit{typ}]\!]}$$

Static semantics rules:

$$\boxed{\text{wf-Unit: Conc-row}} \quad \frac{\begin{array}{ll}\text{(a)} & \mathit{st\text{-}env} \vdash_{\mathit{Conc}} \ \mathit{unm}_1 \text{ has type } [\![[i]\mathit{typ}]\!] \\ \text{(b)} & \mathit{st\text{-}env} \vdash_{\mathit{Conc}} \ \mathit{unm}_2 \text{ has type } [\![[j]\mathit{typ}]\!] \\ \text{(c)} & k = i + j \\ \text{(d)} & \mathit{st\text{-}env} \vdash_{\mathit{Type\text{-}Equality}} \mathit{ty} \equiv [\![[k]\mathit{typ}]\!]\end{array}}{\mathit{st\text{-}env} \vdash_{\mathit{Unit}} [\![\mathit{unm}_1 \ \textsc{conc} \ \mathit{unm}_2]\!] \text{ has type } \mathit{ty}}$$

$$\boxed{\text{wf-Unit: Conc-str}} \quad \frac{\begin{array}{ll}\text{(a)} & \mathit{st\text{-}env} \vdash_{\mathit{Conc}} \ \mathit{unm}_1 \text{ has type } [\![\ \textsc{string} \ [i]\mathit{typ}]\!] \\ \text{(b)} & \mathit{st\text{-}env} \vdash_{\mathit{Conc}} \ \mathit{unm}_2 \text{ has type } [\![\ \textsc{string} \ [j]\mathit{typ}]\!] \\ \text{(c)} & k = i + j \\ \text{(d)} & \mathit{st\text{-}env} \vdash_{\mathit{Type\text{-}Equality}} \mathit{ty} \equiv [\![\ \textsc{string} \ [k]\mathit{typ}]\!]\end{array}}{\mathit{st\text{-}env} \vdash_{\mathit{Unit}} [\![\mathit{unm}_1 \ \textsc{conc} \ \mathit{unm}_2]\!] \text{ has type } \mathit{ty}}$$

The concatentation operation is well-formed when the two units unm_1 and unm_2 have elements from the same basic type. Rules Conc-ty1 and Conc-str1 enable single types to be coerced into unit-length sequences and strings, respectively.

Conc uses the operation $make\text{-}conc_{[TY1,TY2,OTY]}$ to build an automaton which performs the required concatenation:

Dynamic semantics equations:

$$M\text{-}Unit[\![unm_1 \text{ CONC } unm_2]\!](env) =$$
$$(MU1[0 \mapsto 1] \,||\, MU2[0 \mapsto 2] \,||\, make\text{-}conc_{[TY1,TY2,OTY]})\backslash\{1,2\}$$

where

$$MU1 = M\text{-}Unit[\![unm_1]\!](env)$$
$$MU2 = M\text{-}Unit[\![unm_2]\!](env)$$
$$TY1 = out\text{-}type(MU1)$$
$$TY2 = out\text{-}type(MU2)$$
$$OTY = conc\text{-}obj(TY1, TY2)$$

The operation $make\text{-}conc_{[TY1,TY2,OTY]}$ builds an automaton which has input signature $TY1 \times TY2$ and output signature OTY. When handling row-types, it takes an input (v,w), with $v = \langle a_1,\ldots,a_m\rangle \in TY1$, and $w = \langle b_1,\ldots,b_n\rangle \in TY2$, and outputs the value $\langle a_1,\ldots,a_m,b_1,\ldots,b_n\rangle \in OTY$. When handling string types, it takes an input (v,w), with string $v = tag\ ''\, a_1,\ldots,a_m'' \in TY1$, and string $w = tag\ ''\, b_1,\ldots,b_n'' \in TY2$, and outputs the string $tag\ ''\, a_1,\ldots,a_m,b_1,\ldots,b_n'' \in OTY$. 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:

$make\text{-}conc_{[TY1,TY2,OTY]} : GEN\text{-}IOA$
$make\text{-}conc_{[TY1,TY2,OTY]} = A$
where
$\Sigma_I(A) = \{1 \mapsto TY1, 2 \mapsto TY2\}$
$\Sigma_O(A) = \{0 \mapsto OTY\}$
$ty1 = coerce\text{-}type(TY1, TY2)$
$ty2 = coerce\text{-}type(TY2, TY1)$
$States(A) = \{0\}$
$Initials(A) = \{0\}$

$\forall v{:}TY1, w{:}TY2 \cdot$

$$\frac{\begin{array}{ll} \text{(a)} & OTY \in ROW \\ \text{(b)} & \langle a_1, \ldots, a_i \rangle = coerce\text{-}value_{[TY1,ty1]}(v) \\ \text{(c)} & \langle b_1, \ldots, b_j \rangle = coerce\text{-}value_{[TY2,ty2]}(w) \\ \text{(d)} & z = \langle a_1, \ldots, a_i, b_1, \ldots, b_j \rangle \end{array}}{(0 \xrightarrow{\{1 \mapsto v, 2 \mapsto w, 0 \mapsto z\}} 0) \in Transitions(A)} \; \boxed{\text{make-conc: row}}$$

$$\frac{\begin{array}{ll} \text{(a)} & OTY \in TAG\text{-}STRING \\ \text{(b)} & label\text{-}value(tag, \langle a_1, \ldots, a_i \rangle) = coerce\text{-}value_{[TY1,ty1]}(v) \\ \text{(c)} & label\text{-}value(tag, \langle b_1, \ldots, b_j \rangle) = coerce\text{-}value_{[TY2,ty2]}(w) \\ \text{(d)} & z = label\text{-}value(tag, \langle a_1, \ldots, a_i, b_1, \ldots, b_j \rangle) \end{array}}{(0 \xrightarrow{\{1 \mapsto v, 2 \mapsto w, 0 \mapsto z\}} 0) \in Transitions(A)} \; \boxed{\text{make-conc: str}}$$

pre
$compatible\text{-}conc\text{-}types(TY1, TY2) \wedge$
$OTY = conc\text{-}obj(TY1, TY2)$

The operation $conc\text{-}obj(ty_1, ty_2)$ finds the concatenation-type of two compatible types ty_1, ty_2. It first coerces either type if necessary using *coerce-type* and then performs concatenation, between the resultant pair of rows/strings, using *conc-type*:

$conc\text{-}obj : GEN\text{-}TYPE^2 \rightarrow ROW\text{-}OR\text{-}STRING$

$conc\text{-}obj(obj_1, obj_2) = conc\text{-}type(cty_1, cty_2)$

where

$cty_1 = coerce\text{-}type(obj_1, obj_2)$

$cty_2 = coerce\text{-}type(obj_2, obj_1)$

pre

$compatible\text{-}conc\text{-}types(obj_1, obj_2)$

$conc\text{-}type\ ROW\text{-}OR\text{-}STRING^2 \rightarrow ROW\text{-}OR\text{-}STRING$

$\forall\ cty_1, cty_2 : ROW, typ : GEN\text{-}TYPE \bullet$

$conc\text{-}type(cty_1, cty_2) = tuple\text{-}product(\langle ty_1, \ldots, ty_k \rangle)$

where

$cty_1 = tuple\text{-}product(\langle ty_1, \ldots, ty_m \rangle)$

$cty_2 = tuple\text{-}product(\langle ty_1, \ldots, ty_n \rangle)$

$\forall\ i : \{1, \ldots, k\} \bullet ty_i = typ$

$k = m + n$

$\forall\ cty_1, cty_2, typ : TAG\text{-}STRING, typ : CHAR\text{-}SET \bullet$

$conc\text{-}type(cty_1, cty_2) = label\text{-}type(tag, tuple\text{-}product(\langle ty_1, \ldots, ty_k \rangle))$

where

$cty_1 = label\text{-}type(tag, tuple\text{-}product(\langle ty_1, \ldots, ty_m \rangle))$

$cty_2 = label\text{-}type(tag, tuple\text{-}product(\langle ty_1, \ldots, ty_n \rangle))$

$\forall\ i : \{1, \ldots, k\} \bullet ty_i = typ$

$k = m + n$

pre

$compatible\text{-}conc\text{-}types(cty_1, cty_2)$

The relation *compatible-conc-types* determines pairs of types which are compatible for concatenation:

$compatible\text{-}conc\text{-}types\ \subseteq (GEN\text{-}TYPE \times GEN\text{-}TYPE)$

$\forall\ obj_1, obj_2 \in ROW\text{-}OR\text{-}STRING \bullet$

$compatible\text{-}conc\text{-}types(obj_1, obj_2)\ \Leftrightarrow\ (elt\text{-}type(obj_1) = elt\text{-}type(obj_2))$

$\forall\ obj_1 \in ROW\text{-}OR\text{-}STRING \bullet$

$compatible\text{-}conc\text{-}types(obj_1, obj_2)\ \Leftrightarrow\ (obj_2 = elt\text{-}type(obj_1))$

$\forall\ obj_2 \in ROW\text{-}OR\text{-}STRING \bullet$

$compatible\text{-}conc\text{-}types(obj_1, obj_2)\ \Leftrightarrow\ (obj_1 = elt\text{-}type(obj_2))$

The operation *coerce-type* coerces elemental types and character-types to unit-rows and unit-strings respectively. The operation *coerce-value* coerces values in a similar way:

$coerce\text{-}type : GEN\text{-}TYPE^2 \to ROW\text{-}OR\text{-}STRING$

$\forall\ ty_2 \in ROW \bullet coerce\text{-}type(ty_1, ty_2) = \langle ty_1 \rangle$

where

$ty_1 = elt\text{-}type(ty_2)$

$\forall\ ty_2 \in TAG\text{-}STRING \bullet coerce\text{-}type(ty_1, ty_2) = label\text{-}type(tag, \langle ty_1 \rangle)$

where

$ty_1 = elt\text{-}type(ty_2)$

$tag = extract\text{-}label(ty_1)$

$\forall\ ty_1, ty_2 \in ROW\text{-}OR\text{-}STRING \bullet coerce\text{-}type(ty_1, ty_2) = ty_1$

where

$ty_1 \neq elt\text{-}type(ty_2)$

$\forall\ ty_2 \in GEN\text{-}TYPE - ROW\text{-}OR\text{-}STRING \bullet coerce\text{-}type(ty_1, ty_2) = ty_1$

pre

$compatible\text{-}conc\text{-}types(ty_1, ty_2)$

$coerce\text{-}value_{[ty1,ty2]} : ty1 \to ty2$

$\forall\ ty2 \in ROW \bullet coerce\text{-}value_{[ty1,ty2]}(v) = \langle v \rangle$

where

$ty1 = elt\text{-}type(ty2)$

$\forall\ ty2 \in TAG\text{-}STRING \bullet coerce\text{-}value_{[ty1,ty2]}(v) = label\text{-}value(tag, \langle v \rangle)$

where

$ty1 = elt\text{-}type(ty2)$

$extract\text{-}tag(v) = tag$

$\forall\ ty2 \in ROW\text{-}OR\text{-}STRING \bullet coerce\text{-}value_{[ty1,ty2]}(v) = v$

where

$ty1 \neq elt\text{-}type(ty2)$

$\forall\ ty2 \in GEN\text{-}TYPE - ROW\text{-}OR\text{-}STRING \bullet coerce\text{-}value_{[ty1,ty2]}(v) = v$

pre

$compatible\text{-}conc\text{-}types(ty1, ty2)$

The operation $elt\text{-}type(ty)$ returns the component type of a row or string:

$elt\text{-}type : ROW\text{-}OR\text{-}STRING \rightarrow GEN\text{-}TYPE$

$\forall\ ty : ROW, typ : GEN\text{-}TYPE \bullet elt\text{-}type(ty) = typ$

where

$ty = tuple\text{-}product(\langle ty_1, \ldots, ty_k \rangle)$

$\forall\ i \in \{1, \ldots, k\} \bullet ty_i = typ$

$\forall\ ty : TAG\text{-}STRING, typ : CHAR\text{-}SET \bullet elt\text{-}type(ty) = label\text{-}type(tag, \langle typ \rangle)$

where

$ty = label\text{-}type(tag, tuple\text{-}product(\langle ty_1, \ldots, ty_k \rangle))$

$\forall\ i \in \{1, \ldots, k\} \bullet ty_i = typ$

5.6.12 Sequence

Structure:

$$\boxed{\textit{Sequence} \quad ::= \qquad (\textit{unit}_1 \ , \ \ldots \ , \ \textit{unit}_{k+1})}$$

The sequencing operation $(\textit{unm}_1 \ , \ \ldots \ , \ \textit{unm}_{k+1})$ returns a tuple value containing component values produced by $\textit{unm}_1, \ \ldots \ , \ \textit{unm}_{k+1}$. 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:

$$\boxed{\text{wf-Unit: Sequence(a)}} \quad \frac{\textit{st-env} \vdash_{\textit{Unit}} \ \textit{unm}_1 \text{ has type } \textit{ty}_1}{\textit{st-env} \vdash_{\textit{Unit}} [\![(\textit{unm}_1)]\!] \text{ has type } \textit{ty}_1}$$

$$\boxed{\text{wf-Unit: Sequence(b)}} \quad \frac{\begin{array}{ll} \text{(a)} & k \geq 1 \\ \text{(b)} & \forall i \in \{1,\ldots,k+1\} \cdot \textit{st-env} \vdash_{\textit{Unit}} \ \textit{unm}_i \text{ has type } \textit{ty}_i \\ \text{(c)} & \textit{ty} = [\![(\textit{ty}_1,\ldots,\textit{ty}_{k+1})]\!] \end{array}}{\textit{st-env} \vdash_{\textit{Unit}} [\![(\textit{unm}_1 \ , \ \ldots \ , \ \textit{unm}_{k+1})]\!] \text{ has type } \textit{ty}}$$

Dynamic semantics equations:

$$\textit{M-Unit}[\![(\textit{unm}_1)]\!](\textit{env}) = \textit{M-Unit}[\![\textit{unm}_1]\!](\textit{env})$$

$$\textit{M-Unit}[\![(\textit{unm}_1 \ , \ \ldots \ , \ \textit{unm}_{k+1})]\!](\textit{env}) =$$

$$(\textit{MU}_1[0 \mapsto 1] \ || \ \ldots \ || \ \textit{MU}_{k+1}[0 \mapsto k+1] \ || \ \textit{make-sequence}_{[TY]}) \backslash \{1,\ldots,k+1\}$$

where

$$\forall i \in \{1,\ldots,k+1\} \cdot \textit{MU}_i = \textit{M-Unit}[\![\textit{unm}_i]\!](\textit{env})$$

$$\textit{OTY} = \textit{tuple-product}(\langle \textit{out-type}(\textit{MU}_1),\ldots,\textit{out-type}(\textit{MU}_{k+1})\rangle)$$

$$k \geq 1$$

The operation $\textit{make-sequence}_{[OTY]}$ constructs the automaton to perform the required sequencing operation; it takes $k+1$ inputs and produces a single output — a sequence containing the $k+1$ inputs:

$make\text{-}sequence_{[TY]} : GEN\text{-}IOA$

$make\text{-}sequence_{[TY]} = A$

where

$tuple\text{-}product(\langle ty_1, \ldots, ty_{k+1} \rangle) = OTY$

$\Sigma_I(A) = \{1 \mapsto ty_1, \ldots, k+1 \mapsto ty_{k+1}\}$

$\Sigma_O(A) = \{0 \mapsto OTY\}$

$States(A) = \{0\}$

$Initials(A) = \{0\}$

$\forall a_1 : ty_1, \ldots, a_{k+1} : ty_{k+1}.$

$$\boxed{\text{make-sequence}}\ \frac{\begin{array}{ll}\text{(a)} & v = \langle a_1, \ldots, a_{k+1} \rangle \\ \text{(b)} & m = \{1 \mapsto a_1, \ldots, k+1 \mapsto a_{k+1}, 0 \mapsto v\}\end{array}}{(0 \xrightarrow{m} 0) \in Transitions(A)}$$

pre

$TY \in ROW$

5.6.13 Extract

Structure:

$$\boxed{Extract \quad ::= \qquad unit \; // \; enum}$$

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:

(a) $tyn \in \mathsf{dom}\, st\text{-}env \;\wedge\; [\![tag_1 \;\&\; tagty]\!] \in newtype\text{-}of(st\text{-}env(tyn))$

(b) $st\text{-}env \vdash_{Unit} \; unm \; \mathsf{has\ type} \; tyn$

wf-Unit: Extract

$$st\text{-}env \vdash_{Unit} [\![unm \; // \; tag_1]\!] \; \mathsf{has\ type} \; tagty$$

Extract uses the operation $extract_{[ITY]}(tag_1)$ to build an automaton 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:

$$M\text{-}Unit[\![unm \; // \; tag_1]\!](env) = (MU[0 \mapsto 1] \; || \; extract_{[ITY]}(tag_1)) \backslash \{1\}$$

where

$$MU = M\text{-}Unit[\![unm]\!](env)$$

$$ITY = out\text{-}type(MU)$$

$extract_{[ITY]} : Tag \to GEN\text{-}IOA$

$extract_{[ITY]}(tag_1) = A$

where

$label\text{-}type(tag_1, OTY) \in ITY$

$\Sigma_I(A) = \{1 \mapsto ITY\}$

$\Sigma_O(A) = \{0 \mapsto OTY\}$

$States(A) = \{0\}$

$Initials(A) = \{0\}$

$\forall v{:}ITY \cdot$

$$\boxed{\text{extract-1}}\ \frac{label\text{-}value(tag_1, w) = v}{(0 \xrightarrow{\{1 \mapsto v, 0 \mapsto w\}} 0) \in Transitions(A)}$$

$\forall v : ITY, x : OTY \cdot$

$$\boxed{\text{extract-2}}\ \frac{\begin{array}{ll}\text{(a)} & label\text{-}value(tag_2, w) = v \\ \text{(b)} & tag_1 \neq tag_2\end{array}}{(0 \xrightarrow{\{1 \mapsto v, 0 \mapsto x\}} 0) \in Transitions(A)}$$

pre

$tag_1 = extract\text{-}label(tag\text{-}ty) \wedge tag\text{-}ty \in ITY$

5.6.14 Instance

Structure:

$$\boxed{\mathit{Instance} \;::=\; \mathit{fnname}\ (\ \mathit{unit}\ )}$$

Static semantics rules:

$$\frac{\begin{array}{ll} \text{(a)} & \mathit{fnname} \in \mathsf{dom}\, \mathit{st\text{-}env} \\ \text{(b)} & \mathit{st\text{-}env}(\mathit{fnname}) = [\![\ \texttt{funtype}\ (\mathit{ty}_1) \rightarrow \mathit{ty}_2]\!] \\ \text{(c)} & \mathit{st\text{-}env} \vdash_{\mathit{Unit}} \mathit{un}\ \textsf{has type}\ \mathit{ty}_3 \\ \text{(d)} & \mathit{st\text{-}env} \vdash_{\mathit{Type\text{-}Equality}}\ \mathit{ty}_1\ \equiv\ \mathit{ty}_3 \end{array}}{\mathit{st\text{-}env} \vdash_{\mathit{Unit}} [\![\mathit{fnname}(\mathit{un})]\!]\ \textsf{has type}\ \ \mathit{ty}_2}\ \boxed{\textsf{wf-Unit : Instance}}$$

Dynamic semantics equations:

$$\mathit{M\text{-}Unit}[\![\mathit{fnname}(\mathit{un})]\!](\mathit{env}) =$$
$$(\mathit{FN}\ ||\ \mathit{UN}[0 \mapsto 1]) \backslash \{1\}$$

where

$$\mathit{FN} = \mathit{multi\text{-}to\text{-}one\text{-}input}(\mathit{env}(\mathit{fnname}))$$
$$\mathit{UN} = \mathit{M\text{-}Unit}[\![\mathit{un}]\!](\mathit{env})$$

The function *multi-to-one-input* : $\mathit{GEN\text{-}IOA} \rightarrow \mathit{GEN\text{-}IOA}$ transforms automata with many inputs into equivalent automata with a single input.

The main reason that this function is needed is that ELLA function objects are defined as having *multiple* inputs, whereas they are used as though a *single* argument is provided.

This clearly relies on the use of numbers in input signatures to indicate the *order* in which the inputs are composed into a single tuple.

$multi\text{-}to\text{-}one\text{-}input : GEN\text{-}IOA \rightarrow GEN\text{-}IOA$

$multi\text{-}to\text{-}one\text{-}input(A) = [\![\, \mathtt{ioa}\ \ states\ i\text{-}sig\ o\text{-}sig\ inits\ trans\text{-}set]\!]$

where

$states = States(A)$

$i\text{-}sig = \{1 \mapsto tuple\text{-}input\text{-}types(\Sigma_I(A))\}$

$o\text{-}sig = \Sigma_O(A)$

$inits = Inits(A)$

$$(s_1 \xrightarrow{m} s_2) \in Trans(A)$$

$\boxed{\text{multi-to-one-input}}$

$$(s_1 \xrightarrow{tuple\text{-}input\text{-}values(m)} s_2) \in trans\text{-}set$$

pre

$\exists\, n \in \mathbb{N} \bullet \mathrm{dom}\, \Sigma_I(A) = \{1, \ldots, n\}$

$tuple\text{-}input\text{-}types : Signature \rightarrow ROW$

$tuple\text{-}input\text{-}types(sig) = tuple\text{-}product(\langle ty_1,\ \ldots\ ,\ ty_n \rangle)$

where

$\mathrm{dom}\ sig = \{1, \ldots, n\}$

$(\forall\, i \in \{1, \ldots, n\} \bullet ty_i = sig(i))$

$tuple\text{-}input\text{-}values : Binding \rightarrow Binding$

$tuple\text{-}input\text{-}values(bind) = \{1 \mapsto \langle v_1,\ \ldots\ ,\ v_n \rangle\}$

where

$\mathrm{dom}\ bind = \{1, \ldots, n\}$

$(\forall\, i \in \{1, \ldots, n\} \bullet v_i = bind(i))$

5.6.15 Signals

Structure:

$$\boxed{\mathit{Signal} \quad ::= \quad \mathit{name}}$$

Static semantics rules:

$$\boxed{\text{wf-Unit : Signal}}\quad \frac{\begin{array}{ll}\text{(a)} & \mathit{sig} \in \mathsf{dom}\, \mathit{st\text{-}env} \\ \text{(a)} & \mathit{ty} = \mathit{vartype\text{-}of}(\mathit{st\text{-}env}(\mathit{sig}))\end{array}}{\mathit{st\text{-}env} \vdash_{\mathit{Unit}} \mathit{sig} \text{ has type } \mathit{ty}}$$

Dynamic semantics equations:

$\mathit{M\text{-}Unit}(\mathit{sig})(\mathit{env}) = \mathit{UN}$

where

$\mathit{UN} = \mathit{extract\text{-}signal\text{-}info}(\mathit{env}(\mathit{sig}))$

The function *extract-signal-info* is used to extract value semantics from the given signal information:

$\mathit{extract\text{-}signal\text{-}info} : \mathit{BlockSig} \rightarrow \mathit{GEN\text{-}IOA}$

$\mathit{extract\text{-}signal\text{-}info}[\![\, \texttt{make-signal}\ \mathit{OBJSIG}]\!] = \mathit{SIG}$

$\mathit{extract\text{-}signal\text{-}info}[\![\, \texttt{let-signal}\ \mathit{SIG}]\!] = \mathit{SIG}$

5.6.16 Block structures

Structure:

$$\boxed{Block \quad ::= \quad \texttt{BEGIN}\ step_1\ \ldots\ step_{k+1}\ \texttt{OUTPUT}\ unit\ \texttt{END}}$$

Static semantics rules:

$$\frac{\begin{array}{ll}\text{(a)} & \vdash_{redefinition} [\![st_1\ \ldots\ st_{k+1}]\!] \\ \text{(b)} & st\text{-}env \vdash_{Step^*} [\![st_1\ \ldots\ st_{k+1}]\!]\ \text{produces}\ new\text{-}st\text{-}env \\ \text{(c)} & new\text{-}st\text{-}env \vdash_{Unit} un\ \ \text{has type}\ \ ty \\ \text{(d)} & new\text{-}st\text{-}env \vdash_{joined} [\![st_1\ \ldots\ st_{k+1}]\!]\end{array}}{st\text{-}env \vdash_{Unit} [\![\ \texttt{BEGIN}\ st_1\ \ldots\ st_{k+1}\ \texttt{OUTPUT}\ un\ \texttt{END}\ ]\!]\ \ \text{has type}\ \ ty}\ \boxed{\text{wf-Unit : Block}}$$

$$\frac{\begin{array}{ll}\text{(a)} & st\text{-}ev_1 = st\text{-}env \\ \text{(b)} & \forall\, i \in \{1,\ldots,k+1\} \bullet st\text{-}ev_i \vdash_{Step} st_i\ \text{produces}\ st\text{-}ev_{i+1} \\ \text{(c)} & new\text{-}st\text{-}env = st\text{-}ev_{k+1}\end{array}}{st\text{-}env \vdash_{Step^*} [\![st_1\ \ldots\ st_{k+1}]\!]\ \text{produces}\ new\text{-}st\text{-}env}\ \boxed{\text{wf-Step sequence}}$$

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.

Dynamic semantics equations:

$$M\text{-}Unit[\![\ \texttt{BEGIN}\ st_1\ \ldots\ st_{k+1}\ \texttt{OUTPUT}\ un\ \texttt{END}\ ]\!](env) \ = \ UN$$

where

$$UN = M\text{-}Unit(un)(new\text{-}env)$$

$$new\text{-}env = M\text{-}Step^*[\![st_1\ \ldots\ st_{k+1}]\!](env)$$

$$M\text{-}Step^*[\![st_1\ \ldots\ st_{k+1}]\!](env) \ = \ new\text{-}env$$

where

$$ev_1 = env$$

$$(\forall\, i \in \{1,\ldots,k\} \bullet ev_{i+1} = M\text{-}Step(st_i)(ev_i))$$

$$ev_{k+1} = new\text{-}env$$

The auxiliary predicate for checking that all locally ‘made’ signals are ‘joined’ is defined as follows:

$$(\_ \vdash_{joined} \_) \subseteq (StaticEnvironment \times Step^*)$$

$$\boxed{\textsf{joined}}\ \frac{\forall\, nm \in (join\text{-}to\text{-}names(step\text{-}seq) \cap \mathsf{dom}\ st\text{-}env) \bullet st\text{-}env(nm) \in LetType}{st\text{-}env \vdash_{joined} step\text{-}seq}$$

The following auxiliary predicate enforces the 'redefinition' scoping property for block mentioned earlier:

$$(\vdash_{redefinition} \_) \subseteq Step^*$$

$$\boxed{\textsf{redefinition}}\ \frac{\begin{array}{ll} \text{(a)} & free\text{-}sigs_1 = \emptyset \\ \text{(b)} & \forall\, i \in \{1, \ldots, k+1\} \bullet \\ & \quad free\text{-}sigs_{i+1} = free\text{-}sigs_i \cup free\text{-}signals(st_i) \wedge \\ & \quad definition\text{-}signal(st_i) \cap free\text{-}sigs_{i+1} = \emptyset \end{array}}{\vdash_{redefinition} [\![st_1 \ \ldots \ st_{k+1}]\!]}$$

5.6.16.1 Step bindings within blocks

Structure:

$$
\boxed{\begin{array}{rcl}
\mathit{Step} & ::= & \mathit{declaration} \mid \mathit{let} \mid \mathit{make} \mid \mathit{join} \\
\mathit{Let} & ::= & \texttt{LET}\ \mathit{signal} = \mathit{unit}\ . \\
\mathit{Make} & ::= & \texttt{MAKE}\ \mathit{fnname} :\ \mathit{signal}\ . \\
\mathit{Join} & ::= & \texttt{JOIN}\ \mathit{unit} \rightarrow \mathit{signal}\ .
\end{array}}
$$

Static semantics rules:

$\forall\ \mathit{decl} \in \mathit{Declaration}\ \bullet$

$$
\boxed{\text{wf-Step : Declaration}}\ \frac{\mathit{st\text{-}env} \vdash_{\mathit{Declaration}} \mathit{decl}\ \text{produces}\ \mathit{new\text{-}st\text{-}env}}{\mathit{st\text{-}env} \vdash_{\mathit{Step}} \mathit{decl}\ \text{produces}\ \mathit{new\text{-}st\text{-}env}}
$$

$$
\boxed{\text{wf-Step : Let}}\ \frac{\begin{array}{ll}
\text{(a)} & \mathit{st\text{-}env} \vdash_{\mathit{Unit}} \mathit{un}\ \ \text{has type}\ \ \mathit{ty} \\
\text{(b)} & \mathit{sig\text{-}ty} = [\![\ \texttt{let-type}\ \mathit{ty}]\!] \\
\text{(c)} & \mathit{new\text{-}st\text{-}env} = \mathit{st\text{-}env}\ \dagger\ \{\mathit{sig} \mapsto \mathit{sig\text{-}ty}\}
\end{array}}{\mathit{st\text{-}env} \vdash_{\mathit{Step}} [\![\ \texttt{LET}\ \mathit{sig} = \mathit{un}.]\!]\ \text{produces}\ \mathit{new\text{-}st\text{-}env}}
$$

$$
\boxed{\text{wf-Step : Make}}\ \frac{\begin{array}{ll}
\text{(a)} & \mathit{fn\text{-}nm} \in \text{dom}\ \ \mathit{st\text{-}env} \\
\text{(b)} & \mathit{st\text{-}env}(\mathit{fn\text{-}nm}) = [\![\ \texttt{funtype}\ (\mathit{ty}_1) \rightarrow \mathit{ty}_2]\!] \\
\text{(c)} & \mathit{sig} \notin \text{dom}\ \ \mathit{st\text{-}env} \\
\text{(d)} & \mathit{sig\text{-}ty} = [\![\ \texttt{make-type}\ \mathit{ty}_1 \mathit{ty}_2]\!] \\
\text{(e)} & \mathit{new\text{-}st\text{-}env} = \mathit{st\text{-}env} \cup \{\mathit{sig} \mapsto \mathit{sig\text{-}ty}\}
\end{array}}{\mathit{st\text{-}env} \vdash_{\mathit{Step}} [\![\ \texttt{MAKE}\ \mathit{fn\text{-}nm} :\ \mathit{sig}.]\!]\ \text{produces}\ \mathit{new\text{-}st\text{-}env}}
$$

$$
\boxed{\text{wf-Step : Join}}\ \frac{\begin{array}{ll}
\text{(a)} & \mathit{sig} \in \text{dom}\ \ \mathit{st\text{-}env} \\
\text{(b)} & \mathit{st\text{-}env}(\mathit{sig}) = [\![\ \texttt{make-type}\ \mathit{ty}_1 \mathit{ty}_2]\!] \\
\text{(c)} & \mathit{st\text{-}env} \vdash_{\mathit{Unit}} \mathit{un}\ \ \text{has type}\ \ \mathit{ty}_1 \\
\text{(d)} & \mathit{sig\text{-}ty} = [\![\ \texttt{let-type}\ \mathit{ty}_2]\!] \\
\text{(e)} & \mathit{new\text{-}st\text{-}env} = \mathit{st\text{-}env}\ \dagger\ \{\mathit{sig} \mapsto \mathit{sig\text{-}ty}\}
\end{array}}{\mathit{st\text{-}env} \vdash_{\mathit{Step}} [\![\ \texttt{JOIN}\ \mathit{un}\ \ \rightarrow\ \ \mathit{sig}.]\!]\ \text{produces}\ \mathit{new\text{-}st\text{-}env}}
$$

Dynamic semantics equations:

$$M\text{-}Step(decl)(env) = M\text{-}Declaration(decl)(env)$$

$$M\text{-}Step[\![\text{LET } sig = un.]\!](env) = env \dagger \{sig \mapsto RES\}$$

where

$$RES = [\![\texttt{let-signal } UN]\!]$$

$$UN = M\text{-}Unit(un)(env)$$

$$M\text{-}Step[\![\text{MAKE } fn\text{-}nm : sig.]\!](env) = env \dagger \{sig \mapsto RES\}$$

where

$$RES = [\![\texttt{make-signal } OBJ\ SIG]\!]$$

$$SIG = channel_{[TY]}(sig)$$

$$TY = out\text{-}type(OBJ)$$

$$OBJ = multi\text{-}to\text{-}one\text{-}input(FN[0 \mapsto sig])$$

$$FN = env(fn\text{-}nm)$$

$$M\text{-}Step[\![\text{JOIN } un \rightarrow sig.]\!](env) = env \dagger \{sig \mapsto RES\}$$

where

$$RES = [\![\texttt{let-signal } REN\text{-}COMP]\!]$$

$$REN\text{-}COMP = COMP[sig \mapsto 0]$$

$$COMP = (OBJ \parallel UN[0 \mapsto 1]) \backslash \{1\}$$

$$env(sig) = [\![\texttt{make-signal } OBJ\ SIG]\!]$$

$$UN = M\text{-}Unit[\![un]\!](env)$$

The primitive automaton constructor $channel_{[TY]}(sig)$ used above is defined as follows:-

$$channel_{[TY]}\colon Signal \rightarrow \textit{GEN-IOA}$$

$$channel_{[TY]}(sig) = C$$

where

$$\begin{array}{lcl} \Sigma_I(C) & = & \{sig \mapsto TY\} \\ \Sigma_O(C) & = & \{0 \mapsto TY\} \\ States(C) & = & \{0\} \\ Inits(C) & = & States(C) \end{array}$$

$$\forall\ v \in TY \bullet$$

$$\boxed{\text{channel}}\ \dfrac{m = \{sig \mapsto v, 0 \mapsto v\}}{(0 \xrightarrow{m} 0) \in Transitions(C)}$$

Although not used immeadiately above, the following function is used elsewhere to extract the name of the signal that is defined by a *Step*:

$$\textit{definition-signal} : Step \rightarrow \mathcal{F}(Name)$$

$$\forall\ decl \in Declaration \bullet$$

$$\quad \textit{definition-signal}(decl) = \emptyset$$

$$\textit{definition-signal}[\![\ \texttt{LET}\ sig = un\ .]\!] = \{sig\}$$

$$\textit{definition-signal}[\![\ \texttt{MAKE}\ fnnm : sig\ .]\!] = \{sig\}$$

$$\textit{definition-signal}[\![\ \texttt{JOIN}\ un \rightarrow sig\ .]\!] = \{sig\}$$

5.6.17 Case analysis

Structure:

$$\begin{array}{rcl} \mathit{Case} & ::= & \texttt{CASE}\ \mathit{unit}_1\ \texttt{OF}\ \mathit{caseitem}_1\ ,\ \ldots\ ,\ \mathit{caseitem}_{k+1}\ \texttt{ELSE}\ \mathit{unit}_2\ \texttt{ESAC} \\ \mathit{Caseitem} & ::= & \mathit{constset} : \mathit{unit} \end{array}$$

Static semantics rules:

$$\dfrac{\begin{array}{ll} \text{(a)} & \mathit{st\text{-}env} \vdash_{\mathit{Unit}} \mathit{ch} \text{ has type } \mathit{ch\text{-}ty} \\ \text{(b)} & \mathit{st\text{-}env} \vdash_{\mathit{Unit}} \mathit{el} \text{ has type } \mathit{ty} \\ \text{(c)} & \forall\, i \in \{1, \cdots, (k+1)\} \bullet \\ & \quad \mathit{st\text{-}env} \vdash_{\mathit{CaseItem}} \mathit{citm}_i \text{ has type } (\mathit{ch\text{-}ty}, \mathit{ty}) \\ \text{(d)} & \forall\, i, j \in \{1, \cdots, (k+1)\} \bullet \\ & \quad (i \neq j) \Rightarrow \mathit{st\text{-}env} \vdash_{\mathit{disjoint}} \mathit{citm}_i \not\sim \mathit{citm}_j \end{array}}{\mathit{st\text{-}env} \vdash_{\mathit{Unit}} [\![\ \texttt{CASE}\ \mathit{ch}\ \texttt{OF}\ \mathit{citm}_1\ ,\ \ldots\ ,\ \mathit{citm}_{k+1}\ \texttt{ELSE}\ \mathit{el}\ \texttt{END}\ ]\!] \text{ has type } \mathit{ty}}\ \boxed{\text{wf-Unit : Case}}$$

$$\dfrac{\begin{array}{ll} \text{(a)} & \mathit{st\text{-}env} \vdash_{\mathit{Unit}} \mathit{cns} \text{ has type } \mathit{cns\text{-}ty} \\ \text{(b)} & \mathit{st\text{-}env} \vdash_{\mathit{Unit}} \mathit{un} \text{ has type } \mathit{un\text{-}ty} \end{array}}{\mathit{st\text{-}env} \vdash_{\mathit{CaseItem}} [\![\mathit{cns} : \mathit{un}]\!] \text{ has type } (\mathit{cns\text{-}ty}, \mathit{un\text{-}ty})}\ \boxed{\text{wf-CaseItem}}$$

Dynamic semantics equations:

$$
\begin{aligned}
&\textit{M-Unit}[\![\ \texttt{CASE}\ ch\ \texttt{OF}\ citm_1\ ,\ \ldots\ ,\ citm_{k+1}\ \texttt{ELSE}\ el\ \texttt{END}\ ]\!](env) = \\
&\quad (CH[0 \mapsto 1]\ \mid\mid\ G_1\ \mid\mid\ \cdots\ \mid\mid\ G_{k+1}\ \mid\mid\ EL)\backslash\{1\} \\
&\qquad \text{where} \\
&\qquad\quad CH = \textit{M-Unit}[\![ch]\!](env) \wedge \\
&\qquad\quad TY = \textit{out-type}(CH) \wedge \\
&\qquad\quad (\forall\ i \in \{1,\ \cdots, (k+1)\} \bullet\ G_i = \textit{M-CaseItem}_{[TY]}[\![citm_i]\!](env)) \wedge \\
&\qquad\quad EL = \textit{M-ElseItem}_{[TY]}[\![citm_1\ ,\ \ldots\ ,\ citm_{k+1}]\!](el)(env)
\end{aligned}
$$

$$
\begin{aligned}
&\textit{M-CaseItem}_{[TY]}[\![cns : un]\!](env) = guard_{[TY]}(set, val) \\
&\quad \text{where} \\
&\qquad set = \textit{M-ConstSet}[\![cns]\!](env), \\
&\qquad val = \textit{M-Unit}[\![un]\!](env)
\end{aligned}
$$

$$
\begin{aligned}
&\textit{M-ElseItem}_{[TY]}[\![citm_1\ ,\ \ldots\ ,\ citm_{k+1}]\!](un)(env) = \\
&\quad guard_{[TY]}(TY - \bigcup\{\ set_i \subseteq TY \mid i \in \{1,\ \cdots, (k+1)\}\ \}, EL) \\
&\qquad \text{where} \\
&\qquad\quad (\forall\ i \in \{1,\ \cdots, (k+1)\} \bullet set_i = \textit{M-ConstSet}(\textit{constset-of}(citm_i))(env)) \wedge \\
&\qquad\quad EL = \textit{M-Unit}(un)(env)
\end{aligned}
$$

The primitive automaton constructor $guard_{[TY]}$ used above is defined as follows:

$$
\begin{aligned}
&guard_{[TY]} : \mathcal{F}(TY) \times \textit{GEN-IOA} \to \textit{GEN-IOA} \\
&guard_{[TY]}(set, ioa) = G \\
&\quad \text{where} \\
&\qquad \Sigma_I(G) = \{1 \mapsto TY\} \cup \Sigma_I(ioa) \\
&\qquad \Sigma_O(G) = \Sigma_O(ioa) \\
&\qquad \textit{States}(G) = \textit{States}(ioa) \\
&\qquad \textit{Initials}(G) = \textit{Initials}(ioa) \\
&\qquad \forall\ v \in TY, m \in \textit{MapType}(\Sigma_{IO}(ioa)) \bullet
\end{aligned}
$$

$$
\boxed{\text{guard : Trans}}\ \frac{\begin{array}{ll} \text{(a)} & (s_1 \xrightarrow{m} s_2) \in \textit{Transitions}(ioa) \\ \text{(b)} & (v \in set) \\ \text{(c)} & m_1 = \{1 \mapsto v\} \cup m \end{array}}{(s_1 \xrightarrow{m_1} s_2) \in \textit{Transitions}(G)}
$$

The auxiliary case disjointness constraint predicate is defined in terms of a predicate stating what matching does mean:

$(\_ \vdash_{disjoint} \_ \not\sim \_) \subseteq (StaticEnvironment \times CaseItem \times CaseItem)$

$$\boxed{\text{disjoint}} \; \frac{\neg\,(\textit{st-env} \vdash_{matches} cns_1 \sim cns_2)}{\textit{st-env} \vdash_{disjoint} [\![cns_1 : un_1]\!] \not\sim [\![cns_2 : un_2]\!]}$$

$(\_ \vdash_{matches} \_ \sim \_) \subseteq (StaticEnvironment \times ConstSet \times ConstSet)$

$$\boxed{\text{matches : identity}} \; \frac{\text{true}}{\textit{st-env} \vdash_{matches} cns \sim cns}$$

$$\boxed{\text{matches : symmetric}} \; \frac{\textit{st-env} \vdash_{matches} cns_2 \sim cns_1}{\textit{st-env} \vdash_{matches} cns_1 \sim cns_2}$$

$$\boxed{\text{matches : ConstSetAlt}} \; \frac{\exists\, i \in \{1, \ldots, j\} \bullet (\textit{st-env} \vdash_{matches} cns_i \sim cns)}{\textit{st-env} \vdash_{matches} [\![cns_1 \mid \;\cdots\; \mid cns_j]\!] \sim cns}$$

$$\boxed{\text{matches : ConstSetString}} \; \frac{\begin{array}{ll}\text{(a)} & \forall\, i \in \{1, \ldots, n\} \bullet acns_i = cns_1 \\ \text{(b)} & \textit{st-env} \vdash_{matches} [\![(acns_1, \;\cdots\;, acns_n)]\!] \sim cns\end{array}}{\textit{st-env} \vdash_{matches} [\![\, \texttt{string}\; [n] cns_1]\!] \sim cns}$$

$$\boxed{\text{matches : ConstSetRow}} \; \frac{\forall\, i \in \{1, \ldots, n\} \bullet \textit{st-env} \vdash_{matches} acns_i \sim bcns_i}{\textit{st-env} \vdash_{matches} [\![(acns_1, \;\cdots\;, acns_n)]\!] \sim [\![(bcns_1, \;\cdots\;, bcns_n)]\!]}$$

$$\boxed{\text{matches : TypeName (a)}} \; \frac{\begin{array}{ll}\text{(a)} & \textit{st-env} \vdash_{unit} enum \text{ has type } ty \\ \text{(b)} & \textit{st-env} \vdash_{TypeEquality} ty \equiv tynm\end{array}}{\textit{st-env} \vdash_{matches} tynm \sim enum}$$

$$\boxed{\text{matches : TypeName (b)}} \; \frac{\begin{array}{ll}\text{(a)} & tynm_1 \in \text{dom } \textit{st-env} \\ \text{(b)} & [\![tag \;\&\; tynm_2]\!] \in \textit{newtype-parts-of}(\textit{st-env}(tynm_1))\end{array}}{\textit{st-env} \vdash_{matches} tynm_1 \sim [\![tag \;\&\; tynm_2]\!]}$$

5.6.17.1 ConstSets – Case Pattern Matching

Structure:

$$
\begin{array}{rcl}
\mathit{ConstSet} & ::= & \mathit{enumerated} \mid \mathit{constsetalt} \mid \mathit{constsetstring} \\
& \mid & \mathit{constsetrow} \mid \mathit{constsetassoc} \mid \mathit{constsettype} \\
\mathit{ConstSetAlt} & ::= & (\mathit{constset}_1 \,\underline{|}\, \ldots \,\underline{|}\, \mathit{constset}_{k+1}) \\
\mathit{ConstSetString} & ::= & \mathtt{STRING}\ [\mathit{nat}]\mathit{constset} \\
\mathit{ConstSetRow} & ::= & (\mathit{constset}_1\ ,\ \ldots\ ,\ \mathit{constset}_{k+1}) \\
\mathit{ConstSetAssoc} & ::= & \mathit{enum}\ \&\ \mathit{constset} \\
\mathit{ConstSetType} & ::= & \mathit{typename}
\end{array}
$$

Static semantics rules:

Let $en \in \mathit{Enumerated}$,

$$
\boxed{\text{wf-ConstSet : Enumerated}}\ \frac{\mathit{st\text{-}env} \vdash_{\mathit{Enumerated}} en \text{ has type } ty}{\mathit{st\text{-}env} \vdash_{\mathit{ConstSet}} en \text{ has type } ty}
$$

$$
\boxed{\text{wf-ConstSet : ConstSetAlt}}\ \frac{\forall\, i \in \{1,\ldots,k+1\} \bullet \mathit{st\text{-}env} \vdash_{\mathit{ConstSet}} cns_i \text{ has type } ty}{\mathit{st\text{-}env} \vdash_{\mathit{ConstSet}} [\![(cns_1 \,\underline{|}\, \cdots \,\underline{|}\, cns_{k+1})]\!] \text{ has type } ty}
$$

$$
\boxed{\text{wf-ConstSet : ConstSetString}}\ \frac{\begin{array}{ll}\text{(a)} & n \geq 1 \\ \text{(b)} & cns \in \mathit{Enumerated} \\ \text{(c)} & \mathit{st\text{-}env} \vdash_{\mathit{is\text{-}char\text{-}value}} cns\end{array}}{\mathit{st\text{-}env} \vdash_{\mathit{ConstSet}} [\![\ \mathtt{STRING}\ [n]cns]\!] \text{ has type } ty}
$$

$$
\boxed{\text{wf-ConstSet : ConstSetRow}}\ \frac{\forall\, i \in \{1,\ldots,k+1\} \bullet \mathit{st\text{-}env} \vdash_{\mathit{ConstSet}} cns_i \text{ has type } ty_i}{\mathit{st\text{-}env} \vdash_{\mathit{ConstSet}} [\![(cns_1\ ,\ \cdots\ ,\ cns_{k+1})]\!] \text{ has type } [\![(ty_1,\ \cdots\ ,ty_{k+1})]\!]}
$$

$$
\boxed{\text{wf-ConstSet : ConstSetAssoc}}\ \frac{\begin{array}{ll}\text{(a)} & tynm \in \mathrm{dom}\, \mathit{st\text{-}env} \\ \text{(b)} & \mathit{st\text{-}env} \vdash_{\mathit{ConstSet}} cns \text{ has type } ty \\ \text{(c)} & [\![tag\ \&\ ty]\!] \in \mathit{newtype\text{-}of}(\mathit{st\text{-}env}(tynm))\end{array}}{\mathit{st\text{-}env} \vdash_{\mathit{ConstSet}} [\![tag\ \&\ cns]\!] \text{ has type } tynm}
$$

$$\cfrac{\text{(a)}\quad tynm \in Typename \qquad \text{(b)}\quad st\text{-}env \vdash_{Type} tynm}{st\text{-}env \vdash_{ConstSet} tynm \text{ has type } tynm}\ \boxed{\text{wf-ConstSet : ConstSetType}}$$

Dynamic semantics equations:

$$M\text{-}ConstSet[\![enum]\!](env) = out\text{-}values(M\text{-}Unit[\![enum]\!](env))$$

$$M\text{-}ConstSet[\![(cns_1 \mid \cdots \mid cns_{k+1})]\!](env) = \bigcup\{M\text{-}ConstSet(cns)(env) \mid cns \in cns\text{-}set\}$$

where

$$cns\text{-}set = \{cns_1, \cdots, cns_{k+1}\}$$

$$M\text{-}ConstSet[\![\ \texttt{STRING}\ [n]cnst]\!](env) = out\text{-}values(M\text{-}Unit[\![\ \texttt{STRING}\ [n]cnst]\!](env))$$

$$M\text{-}ConstSet[\![(cns_1, \cdots, cns_{k+1})]\!](env) =$$

$$tuple\text{-}product(\langle set_1, \cdots, set_{k+1}\rangle)$$

where

$$\forall\ i \in \{1, \cdots, (k+1)\} \bullet$$

$$set_i = M\text{-}ConstSet[\![cns_i]\!](env)$$

$$M\text{-}ConstSet[\![tag\ \&\ tynm]\!](env) = \{v \mid v \in (TY \cap TAG\text{-}VALUE) \wedge extract\text{-}tag(v) = tag\}$$

where

$$TY = M\text{-}Type(tynm)(env)$$

$$M\text{-}ConstSet(tynm)(env) = M\text{-}Type(tynm)(env)$$

5.7 Function Declarations

Structure:

$$\boxed{\mathit{FnDecl} \quad ::= \quad \mathit{unitfnd} \mid \mathit{primfnd} \mid \mathit{ramfnd} \mid \mathit{reformfnd} \mid \mathit{biopfnd}}$$

Function declarations are divided into five groups; each construct is basic, since it contains all the information necessary to determine its semantics. 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 other four groups the function body is special — the number of inputs is determined by the type of primitive, and 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. For `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'.

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

$$(\_ \vdash_{\mathit{FnDecl}} \_) \subseteq (\mathit{StaticEnvironment} \times \mathit{FnDecl})$$
$$\mathit{M\text{-}FnDecl} : \mathit{FnDecl} \times \mathit{Environment} \to \mathit{GEN\text{-}IOA}$$

5.7.1 UnitFnD

Structure:

$$\boxed{\mathit{UnitFnD} \;::=\; \texttt{FN}\,\mathit{fnname} = (\mathit{signaldecl}_1,\ \cdots\ ,\mathit{signaldecl}_{k+1}) \rightarrow \mathit{type} : \mathit{unit}.}$$

The function declaration for units $[\![\ \texttt{FN}\,\mathit{fnnm} = (\mathit{sd}_1,\ \cdots\ ,\mathit{sd}_{k+1}) \rightarrow \mathit{ty} : \mathit{unm}.]\!]$ 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:

$$\text{wf-FnDecl: UnitFnd}\quad \frac{\begin{array}{ll}
\text{(a)} & \mathit{fnnm} \notin \mathsf{dom}\,\mathit{st\text{-}env} \\
\text{(b)} & \mathsf{card}\,\mathit{extract\text{-}fndecl\text{-}signals}[\![\mathit{sd}_1,\ldots,\mathit{sd}_{k+1}]\!] = k+1 \\
\text{(c)} & \mathit{fnd\text{-}st\text{-}env} = \mathit{extract\text{-}sigdecl\text{-}binding}[\![\mathit{sd}_1,\ldots,\mathit{sd}_{k+1}]\!] \\
\text{(d)} & \mathit{new\text{-}st\text{-}env} = \mathit{st\text{-}env} \dagger \mathit{fnd\text{-}st\text{-}env} \\
\text{(e)} & \mathit{new\text{-}st\text{-}env} \vdash_{\mathit{Unit}} \mathit{unm}\ \mathsf{has\ type}\ \mathit{ty} \\
\text{(f)} & \mathit{ty\text{-}seq} = \mathit{extract\text{-}sigdecl\text{-}types}[\![\mathit{sd}_1,\ldots,\mathit{sd}_{k+1}]\!] \\
\text{(g)} & \mathit{unitfnd\text{-}st\text{-}env} = \mathit{st\text{-}env} \dagger \{\mathit{fnm} \mapsto [\![\ \texttt{funtype}\ (\mathit{ty\text{-}seq}) \rightarrow \mathit{ty}]\!]\}
\end{array}}{\mathit{st\text{-}env} \vdash_{\mathit{FnDecl}} [\![\ \texttt{FN}\,\mathit{fnnm} = (\mathit{sd}_1,\ \cdots\ ,\mathit{sd}_{k+1}) \rightarrow \mathit{ty} : \mathit{unm}.]\!]\ \mathsf{produces}\ \mathit{unitfnd\text{-}st\text{-}env}}$$

Unit function declaration is well-formed when:

1. the function name *fnnm* being introduced has not already been defined.
2. the input signal names are unique,
3. the free variable declarations in the function body *unm* exactly match those in the signal declaration list,
4. the output type of *unm* is the same as the function declaration output.

Unit function declaration first creates automata to represent the semantics of the input signal names and adds them to the (local) environment. The semantics of *unm* is then found, using this environment containing the local input signals. Finally, the input signal names in the function body unit are replaced with the standard natural number names (i.e. 1,2,...), and the resulting object is stored in the *original* environment, accessed by the function name:

Dynamic semantics equations:

$$\textit{M-FnDecl}[\![\ \texttt{FN}\ \textit{fnnm} = (sd_1,\ \cdots\ ,sd_{k+1}) \rightarrow ty : unm.]\!](env) = env \dagger \{\textit{fnnm} \mapsto MU\}$$

where

$$MU = (\textit{M-Unit}[\![unm]\!](\textit{new-env}))[\textit{ren-inputs}(sd_1,\ldots,sd_{k+1})]$$

$$\textit{new-env} = env \dagger \textit{local-env}$$

$$\textit{local-env} = \textit{M-LocalVariables}(\textit{local-binding})(env)$$

$$\textit{local-binding} = \textit{extract-sigdecl-binding}[\![sd_1,\ldots,sd_{k+1}]\!]$$

The semantic function *M-LocalVariables* creates basic automata to represent the semantics of the local variables or signals:

Dynamic semantics equations:

$$\begin{aligned}
&\textit{M-LocalVariables}(\textit{lcl-bnd})(env) = \\
&\quad \{nm \mapsto \textit{channel}_{[TY]}(nm)\ | \\
&\qquad nm \in \mathsf{dom}\,\textit{lcl-bnd}, TY \in \textit{GEN-TYPE} \bullet \\
&\qquad\quad TY = \textit{M-Type}(\textit{vartype-of}(\textit{lcl-bnd}(nm)))(env) \\
&\quad \}
\end{aligned}$$

The operation *extract-sigdecl-binding* finds the types of the input signal names, and returns a static environment storing the variable-types:

$$\textit{extract-sigdecl-binding} : \textit{SigDecl}^* \rightarrow \textit{StaticEnvironment}$$

$$\textit{extract-sigdecl-binding}[\![sigdec_1,\ldots,sigdec_k]\!] = \{nm_1 \mapsto vt_1,\ldots,nm_k \mapsto vt_k\}$$

where

$$\forall i \in \{1,\ldots,k\} \cdot [\![ty_i : nm_i]\!] = sigdec_i \wedge vt_i = [\![\ \texttt{let-type}\ ty_i]\!]$$

Input renaming is performed by the operation *ren-inputs* which creates a mapping from the unit input signalnames to natural number names:

$$\textit{ren-inputs} : \textit{SigDecl}^* \rightarrow \textit{Name} \xrightarrow{\mathsf{map}} \mathbb{N}$$

$$\textit{ren-inputs}[\![sd_1,\ldots,sd_k]\!] = \{signame_i \mapsto i \mid [\![ty_i : signame_i]\!] = sd_i \wedge i \in \{1,\ldots,k\}\}$$

The operation *extract-fndecl-signals* returns the set of signal names occuring in a sequence of signal declarations, and *extract-sigdecl-types* returns the sequence of types:

$extract\text{-}fndecl\text{-}signals : SigDecl^* \rightarrow \mathcal{F}(Name)$

$extract\text{-}fndecl\text{-}signals[\![sigdec_1, \ldots, sigdec_k]\!] = \{nm_i \mid [\![ty_i : nm_i]\!] = sigdec_i \wedge i \in \{1, \ldots, k\}\}$

$extract\text{-}sigdecl\text{-}types : SigDecl^* \rightarrow Type$

$extract\text{-}sigdecl\text{-}types[\![sigdec_1, \ldots, sigdec_k]\!] = [\![(ty_1, \ldots, ty_k)]\!]$

where

$\forall\ i \in \{1, \ldots, k\} \bullet [\![ty_i : nm_i]\!] = sigdec_i$

$extract\text{-}sigdecl\text{-}types[\![ty : nm]\!] = ty$

5.7.2 PrimFnD

Structure:

$$\boxed{PrimFnD \quad ::= \qquad \texttt{FN}\, fnname = (type_1) \rightarrow ty : primitive.}$$

The function declaration for primitives $[\![\ \texttt{FN}\, fnm = (ty_1) \rightarrow ty : prim.]\!]$ 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:

$$\frac{\begin{array}{ll}\text{(a)} & fnm \notin \mathsf{dom}\, st\text{-}env \\ \text{(b)} & st\text{-}env \vdash_{Type\text{-}Equality} ty_1 \equiv ty \\ \text{(c)} & st\text{-}env \vdash_{Primitive} prim \ \mathsf{has\ type}\ ty \\ \text{(d)} & new\text{-}st\text{-}env = st\text{-}env \dagger \{fnm \mapsto [\![\ \texttt{funtype}\ (ty_1) \rightarrow ty]\!]\}\end{array}}{st\text{-}env \vdash_{FnDecl} [\![\ \texttt{FN}\, fnm = (ty_1) \rightarrow ty : prim.]\!] \ \mathsf{produces}\ new\text{-}st\text{-}env}\ \boxed{\text{wf-FnDecl: PrimFnD}}$$

Primitive function declaration is well-formed when:

1. the function name *fnm* being introduced has not already been defined.
2. there is a single input with the same type *ty* as the output
3. 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:

$$\begin{array}{l} M\text{-}FnDecl[\![\ \texttt{FN}\, fnm = (ty_1) \rightarrow ty : prim.]\!](env) = \\ \quad env \dagger \{fnm \mapsto M\text{-}Primitive[\![prim]\!](env)\} \end{array}$$

5.7.3 RamFnD

Structure:

$$\boxed{RamFnD \quad ::= \qquad \texttt{FN}\, \mathit{fnname} = (\mathit{type}_1, \ldots, \mathit{type}_4) \rightarrow \mathit{type}:\ \texttt{RAM}\,(\mathit{unit}).}$$

The function declaration for RAM $[\![\ \texttt{FN}\, \mathit{fnnm} = (\mathit{ty}_1, \ldots, \mathit{ty}_4) \rightarrow \mathit{ty}_5 :\ \texttt{RAM}\,(\mathit{unm}).]\!]$ 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:

$$\begin{array}{ll}
\text{(a)} & \mathit{fnnm} \notin \mathsf{dom}\, \mathit{st\text{-}env} \\
\text{(b)} & \mathit{st\text{-}env} \vdash_{\mathit{Type\text{-}Equality}} \mathit{ty}_1 \equiv \mathit{ty} \\
\text{(c)} & \mathit{st\text{-}env} \vdash_{\mathit{is\text{-}num\text{-}type}} \mathit{ty}_2 \\
\text{(d)} & \mathit{st\text{-}env} \vdash_{\mathit{Type\text{-}Equality}} \mathit{ty}_2 \equiv \mathit{ty}_3 \\
\text{(e)} & \mathit{ty}_4 \in \mathsf{dom}\, \mathit{st\text{-}env} \ \wedge\ \mathit{ety} = \mathit{newtype\text{-}of}(\mathit{st\text{-}env}(\mathit{ty}_4)) \\
\text{(f)} & \mathit{ety} \in (\mathit{CharType} \cup \mathit{Tag}) \wedge \mathit{card\text{-}entype}(\mathit{ety}) = 2 \\
\text{(g)} & \mathit{st\text{-}env} \vdash_{\mathit{Type\text{-}Equality}} \mathit{ty}_5 \equiv \mathit{ty} \\
\text{(h)} & \mathit{st\text{-}env} \vdash_{\mathit{Unit}} \mathit{unm} \ \mathsf{has\ type}\ \mathit{ty} \\
\text{(j)} & \mathit{st\text{-}env} \vdash_{\mathit{is\text{-}const\text{-}value}} \mathit{unm} \\
\text{(k)} & \mathit{ty\text{-}seq} = [\![(\mathit{ty}_1, \ldots, \mathit{ty}_4)]\!] \\
\text{(l)} & \mathit{new\text{-}st\text{-}env} = \mathit{st\text{-}env} \dagger \{\mathit{fnm} \mapsto [\![\ \texttt{funtype}\ (\mathit{ty\text{-}seq}) \rightarrow \mathit{ty}_5]\!]\}
\end{array}$$

wf-FnDecl: RamFnD

$$\mathit{st\text{-}env} \vdash_{\mathit{FnDecl}} [\![\ \texttt{FN}\, \mathit{fnnm} = (\mathit{ty}_1, \ldots, \mathit{ty}_4) \rightarrow \mathit{ty}_5 :\ \texttt{RAM}\,(\mathit{unm}).]\!]\ \mathsf{produces}\ \mathit{new\text{-}st\text{-}env}$$

The operation $\mathit{card\text{-}entype}(\mathit{ty})$ returns the number of tags or characters in type *ty*. If $\mathit{ty} \in \mathit{IntType}$ then it returns zero:

$$\begin{array}{l}
\mathit{card\text{-}entype} : \mathit{NewTypeParts} \rightarrow \mathbb{N} \\
\mathit{card\text{-}entype}(\{\mathit{tag}_1,\ \cdots\ \mathit{tag}_k\}) = k \\
\mathit{card\text{-}entype}[\![\mathit{tag}('\mathit{ch}_1\ \underline{=}\ \ \cdots\ \ \underline{=}\ '\mathit{ch}_k)]\!] = k \\
\mathit{card\text{-}entype}(\mathit{typ}) = 0 \\
\qquad \mathsf{where} \\
\qquad\quad \mathit{typ} \in \mathit{IntType}
\end{array}$$

Function declaration for RAM is well-formed when:

1. the function name *fnnm* being introduced has not already been defined.
2. there are four inputs, and the inputs/output have the required types.
3. Unit *unm* has the required type, and produces a literal.

Function declaration for RAM uses the function $\mathit{make\text{-}ram}_{[DTY,ITY,ETY]}$ to create a basic automaton representing its semantics, and then stores this in the environment, accessed by the function name:

Dynamic semantics equations:

$$\mathit{M\text{-}FnDecl}[\![\ \texttt{FN}\ \mathit{fnnm} = (ty_1,\ldots,ty_4) \to ty_5 :\ \texttt{RAM}\ (unm).]\!](env) =$$
$$env \dagger \{fnnm \mapsto MU\}$$

where

$$MC = \mathit{M\text{-}Unit}[\![unm]\!](env)$$
$$DTY = \mathit{out\text{-}type}(MC)$$
$$ITY = \mathit{M\text{-}Type}[\![ty_2]\!](env)$$
$$ETY = env(ty_4)$$
$$MU = \mathit{make\text{-}ram}_{[DTY,ITY,ETY]}(MC)$$

The operation $\mathit{make\text{-}ram}_{[DTY,ITY,ETY]}(c)$ creates a basic automaton to represent a RAM which stores data of type *DTY*, and has read- and write-addresses of type *ITY*. The write-enable has type *ETY* which contains a sequence of two enumeration values. The elements in the RAM are initialised to values produced by the automaton *c*:

$$\mathit{make\text{-}ram}_{[DTY,ITY,ETY]} : \mathit{GEN\text{-}IOA} \to \mathit{GEN\text{-}IOA}$$
$$\mathit{make\text{-}ram}_{[DTY,ITY,ETY]}(C) = A$$

where

$$ETY = \langle elt_1, elt_2 \rangle$$
$$entype = \mathit{extract\text{-}set}(ETY)$$
$$lb = \mathit{min\text{-}int\text{-}rng}(ITY)$$
$$ub = \mathit{max\text{-}int\text{-}rng}(ITY)$$
$$n = ub - lb + 1$$
$$\Sigma_I(A) = \{1 \mapsto DTY, 2 \mapsto ITY, 3 \mapsto ITY, 4 \mapsto entype\}$$
$$\Sigma_O(A) = \{0 \mapsto DTY\}$$
$$States(A) = \{lst \mid lst \in DTY^* \wedge \mathsf{len}\ lst = n\}$$
$$Initials(A) = \{\langle x_1,\ldots,x_n \rangle \mid x_i \in \mathit{out\text{-}values}(C) \wedge i \in \{1,\ldots,n\}\}$$

$\forall\ a_1,\ldots,a_n, iv \in DTY,\ waddr, raddr \in ITY,\ en \in ETY\ \bullet$

(a) $en = elt_1$

(b) $st_1 = [a_1,\ldots,a_n]$

(c) $label\text{-}value(tag, wval) = waddr \wedge label\text{-}value(tag, rval) = raddr$

(d) $w = wval - (lb-1) \wedge r = rval - (lb-1)$

(e) $st_2 = [a_1,\ldots,a_{w\text{-}1}, iv, a_{w+1},\ldots,a_n]$

(f) $ov = st_2(r)$

(g) $m = \{1 \mapsto iv, 2 \mapsto waddr, 3 \mapsto raddr, 4 \mapsto en, 0 \mapsto ov\}$

make-ram-1

$$(st_1 \xrightarrow{m} st_2) \in Transitions(A)$$

(a) $en = elt_2$

(b) $st = [a_1,\ldots,a_n]$

(c) $label\text{-}value(tag, wval) = waddr \wedge label\text{-}value(tag, rval) = raddr$

(d) $w = wval - (lb-1) \wedge r = rval - (lb-1)$

(e) $m = \{1 \mapsto iv, 2 \mapsto waddr, 3 \mapsto raddr, 4 \mapsto en, 0 \mapsto a_r\}$

make-ram-2

$$(st \xrightarrow{m} st) \in Transitions(A)$$

pre

$ITY \in INT\text{-}RANGE$

$ETY \in (TAG \cup TAG\text{-}CHAR)^* \wedge \mathsf{len}\, ETY = 2$

Addresses range over $\{lb,\ldots,ub\}$ and so must be shifted by an amount $lb-1$ to access the sequence of values $[a_1,\ldots,a_n]$ representing the state, where $n = ub - lb + 1$. When $en = elt_1$ then the memory is changed at the write-address, and then the read-address accesses the *updated* store. When $en = elt_2$, then the memory cannot be updated, and so the state does not change — RAM returns the read-address value. The initial state is set to be a sequence of values $x_i \in out\text{-}values(c), i \in \{1,\ldots,n\}$; if c returns a singleton, then a single initial state will be produced, and if c returns ‘ ? ’ then the set of initial states will include the complete set of states.

The operations *min-int-rng* and *max-int-rng* simply return the smallest and largest integers from tagged integer sets.

$TAG\text{-}INT\text{-}RANGE = \{label\text{-}type(tag, intr) \in TAG\text{-}TYPE \mid intr \in INT\text{-}RANGE\}$

$min\text{-}int\text{-}rng : TAG\text{-}INT\text{-}RANGE \to \mathbb{Z}$
$max\text{-}int\text{-}rng : TAG\text{-}INT\text{-}RANGE \to \mathbb{Z}$
  $max\text{-}int\text{-}rng(tag\text{-}int\text{-}set) = ub$
$\wedge$
  $min\text{-}int\text{-}rng(tag\text{-}int\text{-}set) = lb$
    where
      $int\text{-}set = extract\text{-}type(tag\text{-}int\text{-}set)$
      $\{ub,\ lb\} \in int\text{-}set$
      $\forall\ x \in int\text{-}set \bullet lb \leq x \leq ub$

5.7.4 ReformFnD

Structure:

$$ReformFnD \quad ::= \quad \texttt{FN}\, fnname = (type_1, \ldots, type_{k+1}) \rightarrow type : \texttt{REFORM}\,.$$

The Reform function declaration $[\![\ \texttt{FN}\, fnnm = (ty_1, \ldots, ty_{k+1}) \rightarrow ty : \texttt{REFORM}\,.]\!]$ takes a sequence of inputs with types $ty_1, \ldots, ty_{k+1}$ 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) $fnnm \notin \mathsf{dom}\, st\text{-}env$

(b) $flatten\text{-}types[\![(ty_1, \ldots, ty_{k+1})]\!](st\text{-}env) = type\text{-}seq$

(c) $flatten\text{-}types[\![ty]\!](st\text{-}env) = type\text{-}seq$

(d) $ty\text{-}seq = make\text{-}type\text{-}seq[\![(ty_1, \ldots, ty_{k+1})]\!]$

(e) $new\text{-}st\text{-}env = st\text{-}env \dagger \{fnm \mapsto [\![\ \texttt{funtype}\ (ty\text{-}seq) \rightarrow ty]\!]\}$

wf-FnDecl: ReformFnd

$$st\text{-}env \vdash_{FnDecl} [\![\ \texttt{FN}\, fnnm = (ty_1, \ldots, ty_{k+1}) \rightarrow ty : \texttt{REFORM}\,.]\!] \ \mathsf{produces}\ new\text{-}st\text{-}env$$

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.

$$flatten\text{-}types : StaticEnvironment \rightarrow Type \rightarrow (NewType \cup StringType \cup TypeVoid)^*$$

$$\forall tyn : TypeName \cdot flatten\text{-}types(tyn)(st\text{-}env) = flatten\text{-}types(ty)(st\text{-}env)$$

where

$ty = st\text{-}env(tyn)$

$ty \in TypeNaming$

$$\forall tyn : TypeName \cdot flatten\text{-}types(tyn)(st\text{-}env) = \langle newtype\text{-}of(ty) \rangle$$

where

$ty = st\text{-}env(tyn)$

$ty \in TypeFormer$

$\forall ty : TupleType \cdot \mathit{flatten\text{-}types}(ty)(\mathit{st\text{-}env}) =$

$\mathit{flatten\text{-}types}(ty_1)(\mathit{st\text{-}env}) ^\frown \ldots ^\frown \mathit{flatten\text{-}types}(ty_k)(\mathit{st\text{-}env})$

where

$[\![(ty_1, \ldots, ty_k)]\!] = ty$

$\forall ty : PowerType \cdot \mathit{flatten\text{-}types}(ty)(\mathit{st\text{-}env}) = fty_1 ^\frown \ldots ^\frown fty_k$

where

$[\![[k]typ]\!] = ty$

$\forall i \in \{1, \ldots, k\} \cdot fty_i = \mathit{flatten\text{-}types}(typ)(\mathit{st\text{-}env})$

$\forall ty : StringType \cdot \mathit{flatten\text{-}types}(ty)(\mathit{st\text{-}env}) = \langle ty \rangle$

$\forall ty : TypeVoid \cdot \mathit{flatten\text{-}types}(ty)(\mathit{st\text{-}env}) = \langle ty \rangle$

$\mathit{make\text{-}type\text{-}seq} : TupleType \rightarrow Type$

$\mathit{make\text{-}type\text{-}seq}[\![(ty_1)]\!] = ty_1$

$\mathit{make\text{-}type\text{-}seq}[\![(ty_1, \ldots, ty_{k+1})]\!] = [\![(ty_1, \ldots, ty_{k+1})]\!]$

where

$k \geq 1$

Dynamic semantics equations:

$\mathit{M\text{-}FnDecl}[\![\ \texttt{FN}\ fnnm = (ty_1, \ldots, ty_{k+1}) \rightarrow ty\colon\ \texttt{REFORM}\ .]\!](env) =$

$env \dagger \{fnnm \mapsto MU\}$

where

$ITY = \langle \mathit{M\text{-}Type}[\![ty_1]\!](env), \ldots, \mathit{M\text{-}Type}[\![ty_{k+1}]\!](env) \rangle$

$OTY = \mathit{M\text{-}Type}[\![ty]\!](env)$

$MU = \mathit{make\text{-}reform}_{[ITY,OTY]}$

The operation $\mathit{make\text{-}reform}_{[ITY,OTY]}$ produces a basic automaton which has k inputs $v_1, \ldots, v_k$ and output v, where the sequences of atomic values in the flattened inputs/output are the same:

$make\text{-}reform_{[ITY,OTY]} : GEN\text{-}IOA$

$make\text{-}reform_{[ITY,OTY]} = A$

where

$ITY = \langle ty_1, \ldots, ty_k \rangle$

$\Sigma_I(A) = \{1 \mapsto ty_1, \ldots, k \mapsto ty_k\}$

$\Sigma_O(A) = \{0 \mapsto OTY\}$

$States(A) = \{0\}$

$Initials(A) = \{0\}$

$\forall v_1 : ty_1, \ldots, v_k : ty_k, v : ty \cdot$

(a) $flatten\text{-}values(\langle v_1, \ldots, v_k \rangle)(env) = val\text{-}seq$

(b) $flatten\text{-}values(v)(env) = val\text{-}seq$

(c) $m = \{1 \mapsto v_1, \ldots, k \mapsto v_k, 0 \mapsto v\}$

$$\boxed{\text{make-reform}}\ \frac{}{(0 \xrightarrow{m} 0) \in Transitions(A)}$$

The operation *flatten-values* accepts a sequence of (possibly) composite values, and produces the corresponding sequence of atomic values:

$flatten\text{-}values : Environment \rightarrow GEN\text{-}VALUE \rightarrow GEN\text{-}VALUE$

$\forall v : \bigcup TUPLE \cdot$

$flatten\text{-}values(v)(env) =$

$flatten\text{-}values(v_1)(env) ^\frown \ldots ^\frown flatten\text{-}values(v_k)(env)$

where

$\langle v_1, \ldots, v_k \rangle = v$

$\forall v : GEN\text{-}VALUE - \bigcup TUPLE \cdot flatten\text{-}values(v)(env) = \langle v \rangle$

5.7.5 BiopFnD

Structure:

$$\boxed{\mathit{BiopFnD} \quad ::= \qquad \texttt{FN}\, \mathit{fnname} = (\mathit{type}_1, \ldots, \mathit{type}_{k+1}) \rightarrow \mathit{type} \colon \texttt{BIOP}\ \mathit{biopname}.}$$

Information about the built-in operators is contained in an appendix. Further work is required to use this information to present the static and dynamic semantics of biops for this report.

5.8 Closures and Declarations

Structure:

$$\begin{array}{|rcl|} Closure & ::= & declaration_1 \ldots declaration_{k+1} \\ Declaration & ::= & typedecl \mid fndecl \end{array}$$

The *closure* consists of a non-empty sequence of well-formed declarations.

Static semantics rules:

$$\boxed{\text{wf-Closure}}\ \frac{\emptyset \vdash_{Declaration^*} [\![d_1 \ \cdots \ d_{k+1}]\!] \text{ produces } st\text{-}env}{\vdash_{Closure} [\![d_1 \ \cdots \ d_{k+1}]\!]}$$

$$\boxed{\text{wf-Declaration sequence}}\ \frac{\begin{array}{ll} \text{(a)} & st\text{-}ev_1 = st\text{-}env \\ \text{(b)} & \forall\, j \in \{1, \ldots, k\text{-}1\} \bullet \\ & \quad st\text{-}ev_j \vdash_{Declaration} d_j \text{ produces } st\text{-}ev_{j+1} \\ \text{(c)} & new\text{-}st\text{-}env = st\text{-}ev_{k+1} \end{array}}{st\text{-}env \vdash_{Declaration^*} [\![d_1 \ \cdots \ d_k]\!] \text{ produces } new\text{-}st\text{-}env}$$

$$\boxed{\text{wf-Declaration : FnDecl}}\ \frac{\begin{array}{ll} \text{(a)} & fndecl \in FnDecl \\ \text{(b)} & st\text{-}env \vdash_{FnDecl} fndecl \text{ produces } new\text{-}st\text{-}env \end{array}}{st\text{-}env \vdash_{Declaration} fndecl \text{ produces } new\text{-}st\text{-}env}$$

$$\boxed{\text{wf-Declaration : TypeDecl}}\ \frac{\begin{array}{ll} \text{(a)} & tydecl \in TypeDecl \\ \text{(b)} & st\text{-}env \vdash_{TypeDecl} tydecl \text{ produces } new\text{-}st\text{-}env \end{array}}{st\text{-}env \vdash_{Declaration} tydecl \text{ produces } new\text{-}st\text{-}env}$$

A closure is well-formed precisely when:

1. Each of the declarations are well-formed, within the environment produced by processing earlier declarations.

Dynamic semantics equations:

$$
\begin{array}{l}
\textit{M-Closure}[\![d_1 \ \cdots \ d_{k+1}]\!] = \\
\quad \textit{M-Declaration}^*[\![d_1 \ \cdots \ d_i]\!](\textit{empty-env}) \\
\qquad \text{where} \\
\qquad\quad \textit{empty-env} = \emptyset \\
\qquad\quad (i \in \{1, \ldots, k+1\}) \wedge (d_i \in \textit{FnDecl}) \\
\qquad\quad (\forall\, j \in \{i+1, \ldots, k+1\} \bullet d_j \notin \textit{FnDecl})
\end{array}
$$

$$
\begin{array}{l}
\forall\, \textit{fndecl} \in \textit{Fndecl} \bullet \\
\quad \textit{M-Declaration}(\textit{fndecl})(\textit{env}) = \textit{M-Fndecl}(\textit{fndecl})(\textit{env})
\end{array}
$$

$$
\begin{array}{l}
\forall\, \textit{tydecl} \in \textit{Typedecl} \bullet \\
\quad \textit{M-Declaration}(\textit{tydecl})(\textit{env}) = \textit{M-Typedecl}(\textit{tydecl})(\textit{env})
\end{array}
$$

$$
\begin{array}{l}
\textit{M-Declaration}^*\langle \textit{dcl}_1, \ \cdots \ , \textit{dcl}_n \rangle(\textit{env}) = \textit{new-env} \\
\quad \text{where} \\
\qquad \textit{env}_1 = \textit{env} \\
\qquad (\forall\, i \in \{1, \ldots, n\} \bullet \textit{env}_{i+1} = \textit{M-Declaration}(\textit{dcl}_i)(\textit{env}_i)) \\
\qquad \textit{new-env} = \textit{env}_{n+1}
\end{array}
$$

The semantics of the closure is the environment produced by elaborating each of the declarations in turn, beginning with the empty environment. Notice that the semantics ignores any declarations appearing after the final function declaration.

6 Summary and Further Work

This document presents a framework for a compositional semantics of Core ELLA.

The static semantics 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. 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.

The dynamic semantics is defined in terms of inductively defined partial functions which associate well-formed phrases with IO automata as their denotations. Behavioural equivalence is naturally taken as *trace equivalence*, corresponding to equality of the IO trace languages accepted by the IO automata.

The semantic definitions are in fact presented twice in this document — first with a horizontal organisation and then again with a vertical organisation (see following appendices). The horizontal presentation emphasises the compositional structure of each semantic definition, as governed by the syntactic structure of Core ELLA. The vertical presentation collects each semantic definition into one place, so aiding the comparison of the semantic characteristics of different constructs.

The present formulation of the dynamic semantics make use of a number of semantic primitives, where the transition relations are given by classical set-theoretical constructions. This 'constructive' approach builds particular automata rather than directly defining the IO languages that the primitives accept. Such an approach is useful for constructing 'compilation' tools to compute semantic representations.

A more algebraic approach would concentrate on combinators acting upon IO languages which provide the means for defining new languages in terms of given languages. Such work would then provide a simpler algebraic characterisation of Core ELLA semantic primitives in terms of the IO languages that they accept.

7 Acknowledgements

We would like to thank John Morison and Mike Hill of DRA (RSRE, Malvern) for their valuable input to this paper.

8 References

References

[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.

[BGMW92] H. Barringer, G. Gough, B. Monahan, and A. Williams. A semantic framework for kernel ella. Technical Report D2.3a, Formal Verification Support for Ella, IED project 4/1/1357, September 1992.

[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.

[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.

[Mil90] R. Milner. *Concurrency and Communication*. Prentice Hall, 1990.

[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.

[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.

A Core ELLA : Composite Syntax

$$
\begin{array}{rcl}
\textit{Closure} & ::= & \textit{declaration}_1 \ldots \textit{declaration}_{k+1} \\
\textit{Declaration} & ::= & \textit{typedecl} \mid \textit{fndecl}
\end{array}
$$

$$
\begin{array}{rcl}
\textit{TypeDecl} & ::= & \textit{typenaming} \mid \textit{typeformer} \\
\textit{TypeNaming} & ::= & \texttt{TYPE}\ \textit{typename} = \textit{type} \\
\textit{TypeFormer} & ::= & \texttt{TYPE}\ \textit{typename} = \ \texttt{NEW}\ \textit{newtype}
\end{array}
$$

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

$$
\begin{array}{rcl}
\textit{NewType} & ::= & \textit{enumtype} \mid \textit{assoctype} \mid \textit{inttype} \mid \textit{chartype} \\
\textit{AssocType} & ::= & (\textit{tagtype}_1 \underline{|} \ldots \underline{|} \textit{tagtype}_{k+1}) \\
\textit{IntType} & ::= & \textit{tag}/(\textit{int}_1 \ \underline{..}\ \textit{int}_2) \\
\textit{CharType} & ::= & \textit{tag}(\ '\textit{char}_1 \underline{|} \ldots \underline{|}\ '\textit{char}_{k+1}) \\
\textit{TagType} & ::= & \textit{tag} \mid \textit{tag}\ \&\ \textit{type}
\end{array}
$$

$$
\begin{array}{rcl}
\textit{FnDecl} & ::= & \textit{unitfnd} \mid \textit{primfnd} \mid \textit{ramfnd} \mid \textit{reformfnd} \mid \textit{biopfnd} \\
\textit{UnitFnD} & ::= & \texttt{FN}\ \textit{fnname} = (\textit{signaldecl}_1,\ \cdots\ , \textit{signaldecl}_{k+1}) \rightarrow \textit{type} : \textit{unit}. \\
\textit{PrimFnD} & ::= & \texttt{FN}\ \textit{fnname} = (\textit{type}_1) \rightarrow \textit{type} : \textit{primitive}. \\
\textit{RamFnD} & ::= & \texttt{FN}\ \textit{fnname} = (\textit{type}_1, \ldots, \textit{type}_4) \rightarrow \textit{type} :\ \texttt{RAM}\ (\textit{unit}). \\
\textit{ReformFnD} & ::= & \texttt{FN}\ \textit{fnname} = (\textit{type}_1, \ldots, \textit{type}_{k+1}) \rightarrow \textit{type} :\ \texttt{REFORM}\ . \\
\textit{BiopFnD} & ::= & \texttt{FN}\ \textit{fnname} = (\textit{type}_1, \ldots, \textit{type}_{k+1}) \rightarrow \textit{type} :\ \texttt{BIOP}\ \textit{biopname}. \\
\textit{SignalDecl} & ::= & \textit{type} : \ \textit{signal}
\end{array}
$$

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

$$
\begin{array}{rcl}
\mathit{Unitvoid} & ::= & (\ ) \\
\mathit{Assoc} & ::= & \mathit{enum}\ \&\ \mathit{unit} \\
\mathit{Unitstr} & ::= & \texttt{STRING}\ [\mathit{nat}]\mathit{unit} \\
\mathit{Replicate} & ::= & [\ \mathit{nat}\ ]\ \mathit{unit} \\
\mathit{Query} & ::= & ?\ \mathit{type} \\
\mathit{Index} & ::= & \mathit{unit}\ [\ \mathit{nat}\ ] \\
\mathit{Dyindex} & ::= & \mathit{unit}_1\ [[\ \mathit{unit}_2\ ]] \\
\mathit{Replace} & ::= & \texttt{REPLACE}\ (\mathit{unit}_1, \mathit{unit}_2, \mathit{unit}_3) \\
\mathit{Trim} & ::= & \mathit{unit}\ [\ \mathit{nat}_1\ \underline{..}\ \mathit{nat}_2\ ] \\
\mathit{Conc} & ::= & \mathit{unit}_1\ \texttt{CONC}\ \mathit{unit}_2 \\
\mathit{Sequence} & ::= & (\mathit{unit}_1\ ,\ \ldots\ ,\ \mathit{unit}_{k+1}) \\
\mathit{Extract} & ::= & \mathit{unit}\ //\ \mathit{enum} \\
\mathit{Signal} & ::= & \mathit{name} \\
\mathit{Instance} & ::= & \mathit{fnname}\ (\ \mathit{unit}\ ) \\
\mathit{Block} & ::= & \texttt{BEGIN}\ \mathit{step}_1 \ldots \mathit{step}_{k+1}\ \texttt{OUTPUT}\ \mathit{unit}\ \texttt{END}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{Case} & ::= & \texttt{CASE}\ \mathit{unit}_1\ \texttt{OF}\ \mathit{caseitem}_1\ ,\ \ldots\ ,\ \mathit{caseitem}_{k+1}\ \texttt{ELSE}\ \mathit{unit}_2\ \texttt{ESAC} \\
\mathit{Caseitem} & ::= & \mathit{constset} : \mathit{unit} \\
\mathit{Step} & ::= & \mathit{declaration} \mid \mathit{let} \mid \mathit{make} \mid \mathit{join} \\
\mathit{Let} & ::= & \texttt{LET}\ \mathit{signal} = \mathit{unit}. \\
\mathit{Make} & ::= & \texttt{MAKE}\ \mathit{fnname} : \mathit{signal}. \\
\mathit{Join} & ::= & \texttt{JOIN}\ \mathit{unit} \rightarrow \mathit{signal}.
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{ConstSet} & ::= & \mathit{enumerated} \mid \mathit{constsetalt} \mid \mathit{constsetstring} \\
 & \mid & \mathit{constsetrow} \mid \mathit{constsetassoc} \mid \mathit{constsettype} \\
\mathit{ConstSetAlt} & ::= & (\mathit{constset}_1\ \underline{|}\ \ldots\ \underline{|}\ \mathit{constset}_{k+1}) \\
\mathit{ConstSetString} & ::= & \texttt{STRING}\ [\mathit{nat}]\mathit{constset} \\
\mathit{ConstSetRow} & ::= & (\mathit{constset}_1\ ,\ \ldots\ ,\ \mathit{constset}_{k+1}) \\
\mathit{ConstSetAssoc} & ::= & \mathit{enum}\ \&\ \mathit{constset} \\
\mathit{ConstSetType} & ::= & \mathit{typename}
\end{array}
$$

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

Primitive	$::=$	*adelay* | *idelay* | *sample*
Adelay	$::=$	DELAY $(unit_1, nat_1, unit_2, nat_2)$
Idelay	$::=$	IDELAY $(unit, nat)$
Sample	$::=$	SAMPLE $(nat_1, unit, nat_2)$

DeclInfo	$::=$	*typedecl* | *funtype* | *vartype*
FunType	$::=$	`funtype` $(type_1) \rightarrow type_2$
SigType	$::=$	*lettype* | *maketype*
LetType	$::=$	`let-type` *type*
MakeType	$::=$	`make-type` $type_1$ $type_2$
Nat	$=$	$\mathbb{N}$
Int	$=$	$\mathbb{Z}$
Tag	$=$	*Name*
Typename	$=$	*Name*
Fnname	$=$	*Name*
Biopname	$=$	*Name*

Name	$=$	**LEXICAL ITEM**
Char	$=$	**LEXICAL ITEM**

B 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:

$$
\boxed{\begin{array}{rcl}
\mathit{BinTree} & ::= & \mathit{nil} \mid \mathit{tip} \mid \mathit{node} \\
\mathit{Nil} & ::= & \texttt{emptytree} \\
\mathit{Tip} & ::= & \texttt{tip}\ \mathit{data} \\
\mathit{Node} & ::= & \texttt{node}\ \mathit{data}\ \texttt{of}\ \mathit{bintree}_1\ \texttt{and}\ \mathit{bintree}_2 \\
\\
\mathit{Data} & ::= & \mathit{nat} \mid \mathit{name} \\
\mathit{Nat} & = & \mathbb{N} \\
\mathit{Name} & = & \textbf{LEXICAL ITEM}
\end{array}}
$$

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:

$$
\begin{array}{l}
\text{inference rule}_1 \\
\ldots \\
\text{inference rule}_k
\end{array}
$$

The static semantic rules above will be discussed here.

Dynamic semantics equations:

$$
\begin{array}{l}
\text{equation}_1 \\
\ldots \\
\text{equation}_n
\end{array}
$$

Dynamic semantics rules:

$$
\begin{array}{l}
\text{inference rule}_1 \\
\ldots \\
\text{inference rule}_m
\end{array}
$$

The dynamic semantic equation and/or rules presented above would then be discussed here.

B.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 consititute 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:

$$\boxed{\textsf{rulename 1}}\;\frac{\textit{premiss}}{\textit{conclusion}}$$

This is essentially an 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:

$$\boxed{\textsf{rulename 2}}\;\frac{\begin{array}{ll}\text{(a)} & \textit{premiss}_1 \\ \dots & \dots \\ \text{(n)} & \textit{premiss}_n\end{array}}{\textit{conclusion}}$$

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:

$$\boxed{\textsf{rulename 3}}\;\begin{array}{c}\textit{premiss} \\ \hline\hline \textit{conclusion}\end{array}$$

B.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 elegent and simple way of structurally analysing data forms. For example, we may have a function defined like this:

$$
\begin{aligned}
& \mathit{sumtree} : \mathit{BinTree} \to \mathbb{N} \\
& \quad \mathit{sumtree}[\![\texttt{emptytree}]\!] = 0 \\
& \quad \mathit{sumtree}[\![\texttt{tip}\ n]\!] = \mathit{num\text{-}of}(n) \\
& \quad \mathit{sumtree}[\![\texttt{node}\ d\ \texttt{of}\ bt_1\ \texttt{and}\ bt_2]\!] = n_1 + n_2 + 2.n_3 \\
& \qquad \text{where} \\
& \qquad\quad n_1 = \mathit{sumtree}(bt_1) \\
& \qquad\quad n_2 = \mathit{num\text{-}of}(d) \\
& \qquad\quad n_3 = \mathit{sumtree}(bt_2)
\end{aligned}
$$

$$
\begin{aligned}
& \mathit{num\text{-}of} : \mathit{Data} \to \mathbb{N} \\
& \quad \forall\ \mathit{nm} \in \mathit{Name}, n \in \mathit{Nat} \bullet \\
& \qquad \mathit{num\text{-}of}(n) = n \\
& \qquad \mathit{num\text{-}of}(\mathit{nm}) = 1
\end{aligned}
$$

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:

$$\mathit{sumtree}[\![\texttt{node}\ \boxed{\text{abc}}\ \texttt{of emptytree and} (\texttt{tip}\ 3)]\!] = 7$$

B.3 The *where* notation

The where clause is 'syntactic sugar' for logical implication and global universal quantification of free variables:

logical-formula

where

logical-formula$_1$

...

logical-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!

B.4 Semantics 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:

$$my\text{-}fun(a,b,c):\ \mathbb{N}\times\mathbb{N}\times\mathbb{N}\to\mathbb{R}$$

pre

$$a+b\text{-}c \neq 0 my\text{-}fun(x,y,z) = \frac{x.y}{x+y\text{-}z}$$

Combined global and local constraints can be specified as follows:

$div\text{-}tree(t) \colon BinTree \to \mathbb{R}$

pre

$t \neq$ `emptytree`

$div\text{-}tree[\![\ \mathtt{tip}\ d]\!] = \frac{2}{d}$

pre

$d \in \mathbb{N} \wedge d > 0$

$div\text{-}tree[\![\ \mathtt{node}\ d\ \mathtt{of}\ bt_1\ \mathtt{and}\ bt_2]\!] = v$

where

$v = 2.d + div\text{-}tree(bt_1)\ div\text{-}tree(bt_2)$

pre

$d \in \mathbb{N} \wedge d > 0$

Any local pre-condition's are simply conjoined (with the global condition) to the other predicates in the context of a where clause.

B.5 Type parameterisation

We have introduced 'generic' functions by allowing type parameters to be used, such as:

$$reverse_{[A]} : A^* \to A^*$$
$$reverse_{[B]}([\,]) = [\,]$$
$$reverse_{[C]}(a :: l) = reverse_{[C]}(l) \frown [a]$$

In some sense, this denotes a parameteric 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.

C Core ELLA : Static Semantics

This will contain all of the static semantics definitions all pulled together in one place.

D Core ELLA : Dynamic Semantics

This will contain all of the dynamic semantics definitions all pulled together in one place.

E Semantics Index