Dependent Vector Types for Data Structuring in Multirate Faust[☆]

Pierre Jouvelot[a,*], Yann Orlarey[b]

[a]*CRI, Mathématiques et systèmes, MINES ParisTech, 35 rue Saint-Honoré, 77305 Fontainebleau, France*
[b]*Grame, 9 Rue du Garet BP 1185, 69202 Lyon Cedex 01, France*

Abstract

Faust is a functional programming language dedicated to the specification of executable monorate synchronous musical applications. To extend Faust capabilities to important domains such as FFT-based, spectral processing, we introduce here a multirate extension of the core Faust language. The novel idea is to link rate changes to data structure manipulation operations. Creating a vector-valued output signal divides the rate of input signals by the vector size, while serializing vectors multiplies rates accordingly. As duals to vectors, we also introduce record-like data structures, which are used to gather data but do not change signal rates. This interplay between data structures and rates is made possible in the language static semantics by the introduction of dependent types. We present a typing semantics, a denotational semantics and correctness theorems that show that this data structuring/multirate extension preserves the language synchonous characteristics. This new design is under implementation in the Faust compiler.

Keywords: Domain specific languages, Audio signal processing, Multirate computing, Dependent type systems, Static semantics, Denotational semantics

[☆]A preliminary version of this paper has been published in the Proceedings of the 7$^{\text{th}}$ International Sound and Music Conference SMC'10, Barcelona, July 2010.

[*]Corresponding author: tel. +33 (1) 6469 4846, fax +33 (1) 6469 4847
Email addresses: `pierre.jouvelot@mines-paristech.fr` (Pierre Jouvelot), `orlarey@grame.fr` (Yann Orlarey)

1. Introduction

Since Music III, the first language for digital audio synthesis, developed by Max Mathews in 1959 at Bell Labs, to Max [1], and from MUSICOMP, considered one of the very first music composition languages, developed by Lejaren Hiller and Robert Baker in 1963, to OpenMusic [2] and Elody [3], research in music programming languages has been very active and innovative. With the convergence of digital arts, such languages, and in particular visual programming languages like Max, have gained an even larger audience, well outside the computer music research community.

Within this context, the Faust language [4] introduces a dual programming paradigm, based on a highly abstract, purely functional approach to signal processing while offering a high level of performance. Faust semantics is based on a clean and sound framework that enables mathematical correction proofs of Faust applications to be performed, while being complementary to current audio languages by providing a viable alternative to C/C++ for the development of efficient signal processing libraries, audio plug-ins or standalone applications.

The definition of the Faust programming language uses a two-tiered approach: (1) a core language provides constructs to manage signal transformations and (2) a macro language is used on top of this kernel to build and manipulate signal processing patterns. The macro language has rather straightforward syntax and semantics, since it is a syntactic variant of the untyped lambda-calculus with a call-by-name semantics (see [5]). On the other hand, core Faust is more unusual, since, in accordance with its musical application domain, it is based on the notion of "signal processors" (see below).

The original definition of Faust provided in [6] is based on monorate signal processors; this is a serious limitation when specifying spectral-based sound manipulation algorithms (such as FFT) or extending the language applicability outside the music domain, for instance for image analysis and manipulation (such as data compression). We propose here a multirate extension of Faust based on a key innovative principle: data rate changes are intertwined with vector data structure manipulation operations, i.e., creating an output signal where samples are vectors divides the rate of input signals by the vector size, while serializing vectors multiplies rates accordingly. We also introduce new, dual constructs to build record-like signals; contrarily to vector operations, record signals do not induce signal rate modifications. Since Faust current definition does not offer first-class structured data, this proposal kills two birds with one stone by adding both multirate processing and data structures; this interplay between vectors, records and rates is made possible in the typing semantics of Faust by the introduction of dependent types.

The contributions of this paper are as follows: (1) the specification of a new extension of Faust for vector processing, record data manipulation and multirate applications, (2) a static typing semantics of Faust, based on dependent types, (3) a denotational semantics of Faust (the one presented in [6] is operational) and (4) Subject Reduction and Rate Correctness theorems that validate the multirate synchronous nature of this vector extension.

After this introduction, Section 2 provides a brief informal overview of Faust basic operations. Section 3 is a proposal for a multirate extension of this core, which we illustrate with a simple vector application implementing a Haar-like subsampling operation. Section 4 defines the static domains used to define Faust static typing semantics (Section 5). Section 6 defines the semantic domains and rules used in the Faust dynamic denotational semantics, which is shown to be compatible with the static semantics in Section 7. Proving that this structuring and multirate extension of Faust indeed behaves properly, i.e., that signals of different rates merge gracefully in a multirate program, is the subject of the Rate Correctness theorem in Section 8. The last section concludes.

2. Overview of Faust

A Faust program does not describe a sound or a group of sounds, but a kind of *signal processor*, something that gets input signals, itself a function from time ticks t to values, and produces output signals. The program source is organized, basically, as a set of definitions mapping identifiers to expressions; the keyword identifier `process` is the equivalent of `main` in C. Running a Faust program amounts to plugging the I/O signals implicity used by `process` to the actual sound environment, such as a microphone or an audio system for instance, usually via software audio card managers such as Jack[1].

To begin with, here are two very simple Faust examples. The first one produces silence, i.e., a signal providing an infinite supply of 0s:

```
process = 0;
```

Note that `0` is an unusual signal processor, since it takes an empty set of input signals and generates a signal of constant values, namely the integer 0. The second

[1] `http://www.jackaudio.org.`

simple example illustrates the conversion of a two-channel stereo input signal into a one-channel mono output signal using the + primitive that adds its two input signals together to yield a single, summed signal:

```
process = +;
```

Faust primitives are assembled via a set of high-level composition operations on signal processors, generalizations of the mathematical function composition operator $\circ$ and defined via a block-diagram algebra [7]. For instance, connecting the output of + to the input of `abs` in order to compute the absolute value of the summed output signal can be specified using the sequential composition operator "`:`" (colon):

```
process = + : abs;
```

Here is an example of parallel composition (think of a stereo cable) using the operator "`,`" that puts in parallel its left and right expressions. This example uses the _ (underscore) primitive that denotes the identity function on signals (akin to a simple audio cable for a sound engineer):

```
process = _,_;
```

These operators can be arbitrarily combined, modulo typing constraints we present below. For example, to multiply a mono, input signal by 0.5, one can write:

```
process = _,0.5 : *;
```

Taking advantage of some syntactic sugar the details of which are not addressed here, the above example can be rewritten, using what functional programmmers know as curryfication:

```
process = *(0.5);
```

The recursive composition operator "`~`" can be used to create processors with delayed cycles. Here is the example of an integrator:

```
process = + ~ _;
```

where the "~" operator connects here in a feedback loop the output of + to the input of "_", via an implicit connection to the `mem` signal processor which implements a 1-sample delay, and the output of "_" is then used as one of the inputs of +. As a whole, `process` thus takes a single input signal s and computes an output signal s' such that[2] $s'(t) = s(t) + s'(t-1)$, thus performing a numerical integration operation.

To illustrate the use of this recursive operator and also provide a more meaningful audio example, the following 3-line Faust program defines a pseudo-noise generator:

```
random   = +(12345) ~ *(1103515245);
noise    = random,2147483647.0 : /;
volumeUI = vslider("noise[style:knob]", 0,0,100,0.1);
process = (noise,volumeUI : *),100 : /;
```

The definition of `random` specifies a (pseudo) random number generator that produces a signal s such that $s(t) = 12345 + 1103515245 * s(t-1)$. Indeed, the expression `+(12345)` denotes the operation of adding `12345` to a signal, and similarly for `*(1103515245)`. These two operations are recursively composed using the ~ operator, which connects in a feedback loop the output of `+(12345)` to the input of `*(1103515245)` (via an implicit 1-sample delay) and the output of `*(1103515245)` to the single input of `+(12345)`.

The definition of `noise` transforms the random signal into a noise signal by scaling it between -1.0 and +1.0, while the definition of `process` adds a simple user interface to control sound production; the noise signal is multiplied by the value delivered by a slider to control its volume. The whole `process` expression thus does not take any input signal but outputs a signal of pseudo random numbers (see the block diagram representation of this `process` in Figure 1, where the little square near the addition block denotes a 1-sample delay operator).

The last two composition operators in the definition of core Faust, <: and :>, perform fan-out and fan-in transformations, as we illustrate in the next section

3. Multirate Extension

Traditional synchronous languages such as Esterel, Lustre, Signal or State Charts [8] are built upon the concept of clocks and time stamps upon which computation steps are, one way or another, scheduled; the presence (or absence) of

[2] $s'(-1)$ is set to 0 by Faust.

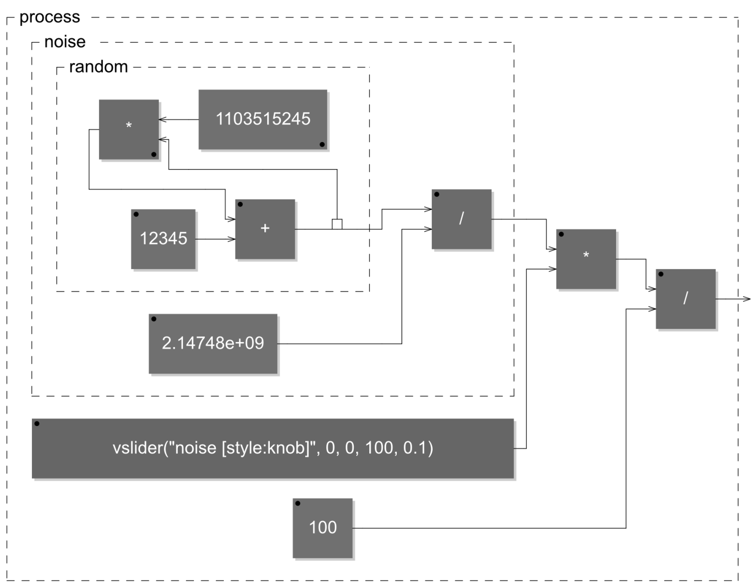

Figure 1: Noise generator `process` block diagram

clock ticks are generally used to activate (or stop) processing. The use of different clocks allows program parts to be activated at different rates. Clocks can be seen as objects of interest either at the programmer's level (e.g., in Lucid Synchrone [9]), at the static semantics level ([10], [11]) or at the mathematical level ([12]).

Faust, as described in [6], is a monorate language; in monorate languages, there is just one time domain involved when accessing successive signal values. However, digital signal processing traditionally relies heavily upon subsampling and oversampling operations, which naturally lead to the introduction of multirate concepts. Since Faust targets the domain of highly efficient, multimedia (mostly audio) DSP processing, we suggest to use simpler, multiple rates to deal with such issues [13], instead of one of the more general clock designs reviewed above. We informally describe below this approach, and illustrates it with a simple example of its use.

3.1. Rates for vector processing

We propose to see clocking issues as an add-on to the Faust static semantics (Faust is a strongly typed language). Rates (or frequencies) f are elements of the $\mathrm{Rate} = \mathbb{Q}$ domain. Signals, which are traditionnally typed according to the type of their codomain, will now be characterized by a pair, called a *rated type*, formed by a type and a rate: $\mathrm{Type}^\sharp = \mathrm{Type} \times \mathrm{Rate}$.

The first key idea is to posit that multiple rates in an application are introduced via vectors. Vectors are created using the new `vectorize` primitive; informally, it collects n consecutive samples (the constant value n is provided by the signal that is the second argument to this primitive) from an input signal of rate f and outputs vectors with n elements at rate f/n; if the input values are of type τ, then output vector samples have type $\mathrm{vector}_n(\tau)$. The dual `serialize` primitive maps a signal of vectors of type $\mathrm{vector}_n(\tau)$ at rate f to the signal of rate $f \times n$ of their linearized elements, of type τ. The primitive `[]` provides, using as inputs a signal of vectors and one of integer indexes, an output signal of successively indexed vector elements. Finally, the primitive[3] `#` builds a signal of concatenated vectors from its two vector signal inputs.

The second key feature of this multirate extension is, as we just saw, that the size n of vectors is encoded into vector types; moreover this size is provided via the *value* of a signal, argument of the `vectorize` primitive. This calls for a dependent-type [14] static semantics that embeds values within types. Since Faust strives for high run-time performance, this type system must furthermore be sophisticated enough to be able to ensure, at compile time, that a given signal is constant (when it is to be used as a signal denoting the size of a vector): we introduce intervals of values in the static semantics to deal with such an issue. Basic rate values are, in practice, provided by the sampling rate of the audio card manager to which a `process` signal processor is eventually linked.

We show below that this interplay between types, vector sizes and rates leads to the addtition of rate constraints to the more traditional typing constraints of Faust static semantics.

3.2. Records and Vectors Rating Duality

Most traditional programming languages offer, at their most fundamental level, at least two kinds of data structuring concepts: vectors and records. There are natural dualities between these notions:

1. Vectors are indexed-based collections of elements, while records are symbol-based (via field names);
2. Vector elements are ordered, while record fields generally are not (some languages such as C support some notion of field ordering in their subtyping

[3] The notation $\sharp$ is, of course, not related to the one used for the domain of rated types.

relationship);

3. Vector sizes are generally dynamic values, while the number of fields in records is a compile-time constant (again, some languages also allow some leeway here, in subtyping or inheritance relationships).

Note that our Faust extension does not enforce this last duality property since, for efficiency reasons, we decided to make vector sizes compile-time constants. However, the presence of rates into the static semantics of our Faust extension suggests to add to these existing duality relationships a new duality relation (rating duality) between vectors and records:

4. Vector (signal) constructs are rate modifiers, while record (signals) are not.

In our proposal, a record constructor signal such as `[foo,bar,baz>` accepts, as inputs, a collection of signals, here three, that operate at the same rate and outputs a single signal, still with the same rate, each of its samples being a labelled collection of input samples. Accessing elements of a record signal is symbol-based: `<bar,foo]` takes as input a signal of records and outputs two signals of the corresponding elements.

Before describing formally our framework in the remainder of this paper, we illustrate it with an example.

3.3. Haar Filtering, an Example

To get a better intuitive understanding of how data structuring constructs interact with Faust primitives, we present a Haar-like downsampling process, which is a simplified step in the Discrete Wavelet Transform shown to be of use, for instance, in some audio feature extraction algorithms [15]. The signal processor `process` takes an input signal s at rate f and produces two output signals at rate $f/2$, the mean o_1 and difference o_2, such that $o_1(t) = (s(2t) + s(2t+1))/2$ and $o_2(t) = o_1(t) - s(2t+1)$. It could be defined in our extended Faust as follows:

```
down     = vectorize(2) : [](1);
mean     = _ <: _,mem :> /(2);
diff = _ <: <mid],<second] : -;
process =
   _ <: (mean:down),down : [mid,second> <: <mid],diff;
```

Here, `down` gathers the data from its input signal in pairs stored in vectors of size 2 (hence the size `2` used in the curried version of `vectorize`) from which the second element is extracted, again using a signal processor, here `[]`, curried over its second argument `1` (vector indices start at 0). This function downsamples its input signal of rate f into an output signal of rate $f/2$, picking one value over two from the input.

The definition of `mean` indicates that its input signal s (denoted by _) is duplicated, using the <: fan-out operator. Two copies are expected since the output of <: is fed into a parallel composition of two one-input signals: the first copy is simply passed along by _, while the second one is being delayed via `mem` by one sample. Both signals $s(t)$ and its delayed copy $s(t-1)$ are then merged, using the fan-in operator :>, which adds the mixed signals to $s(t) + s(t-1)$; this sum signal is then divided by `2` using a curried division operation to yield an average signal $m(t) = (s(t) + s(t-1))/2$.

The signal processor `process` duplicates its single input s (as before, _) to a two-input parallel process: the first copy is averaged using `mean` and then downsampled using sequencing with `down`, yielding signal m_2; the second copy is simply downsampled, yielding s_2. These two parallel data signals are fed into the two-input signal processor `[mid,second>`, which outputs a single signal of record-like structures with two fields, named `mid` and `second`. This record signal is fanned-out into the parallel two-input signal processor `(<mid],diff)`: (1) `<mid]` takes a signal of records, keeping only the component named `mid` and thus retrieving signal m_2 and (2) `diff` fans out its input signal of records to a parallel process that destructures each of them before substraction, yielding $m_2 - s_2$. The end result is the expected pair of signals $(o_1, o_2) = (m_2, m_2 - s_2)$ of downsampled means and differences.

4. Static Domains

The multirate extension of Faust static semantics relies heavily on dependent typing, which is formally defined below.

4.1. Dependent Types

Since the values embedded in signals are typed, the static typing semantics of extended Faust uses basic types b in Base, which is a defined set of predefined types:

$$b \in \mathrm{Base} \quad = \quad \mathrm{int} \mid \mathrm{float}\ .$$

Since our type system uses dependent types, we need a way to abstract values to yield a decidable framework. We introduce spans a in Span, which are pairs of signed integers n (or m); spans represent the intervals of values that expressions may have at run time:

$$\begin{array}{rcl} n, m \in \mathbb{Z}^{\omega} & = & \{-\omega, +\omega\} \cup \mathbb{Z} \ , \\ a \in \mathrm{Span} & = & \mathbb{Z}^{\omega} \times \mathbb{Z}^{\omega} \ , \end{array}$$

where we assume the usual extensions of arithmetic operations on $\mathbb{Z}$ to $\mathbb{Z}^{\omega}$; we take care in the following to avoid introducing meaningless expressions such as $-\omega + +\omega$. Note that we use integer spans here for both integer and floating-point values for simplicity purposes; extending our framework to deal with floating-point spans is straightforward. A span $a = (n, m)$ is written $[n, m]$ in the sequel.

All base-typed expressions are typed with an element b of Base, together with a span $[n, m]$ that specifies an over-approximation of the set of values these expressions might denote. Vectors, as groups of n values, are typed using their size (the number n) and the type of their elements. Records are typed according to their list of field names and the type of the corresponding element in the data record. Finally, since signed integers are part of types, via spans, we need to perform some operations over these values, and thus introduce the notion of type addition. The type domain Type of types τ is then:

$$\begin{array}{rcl} \tau \in \mathrm{Type} & = & \mathrm{Base} \times \mathrm{Span} \mid \\ & & \mathbb{N} \times \mathrm{Type} \mid \\ & & \mathrm{Record} \mid \\ & & \mathrm{Type} \times \mathrm{Type} \ , \\ \\ u \in \mathrm{Record} & = & \displaystyle\bigcup_{n>0} \mathrm{Ide}^n \times \mathrm{Type}^n \ . \end{array}$$

As a shorthand, we note :

- $b[a]$ for base types b with span a;
- $\mathrm{vector}_n(\tau)$ for vector types of n elements of type τ;
- $(\mathtt{L}, \mathtt{T})$ for records, where `L` is a list of field names and `T` a list of corresponding types;
- and $\tau + \tau'$ for the type resulting from performing the addition operation on two types τ and τ'.

Not all combinations of these type-building expressions make sense. We formally define below the notion of a well-formed type:

Definition 1 (Well-Formed Type $\mathit{wff}(\tau)$).
A type τ is well-formed*, noted $\mathit{wff}(\tau)$, iff:*

- *when $\tau = b[n,m]$, then $n \leq m$, $\neg(n = m = -\omega)$ and $\neg(n = m = +\omega)$;*
- *when $\tau = \mathrm{vector}_n(\tau')$, then $\mathit{wff}(\tau')$ and $n \geq 0$;*
- *when $\tau = (\mathtt{L}, \mathtt{T})$, then $\mathit{wff}(\tau')$ for all τ' in $\mathtt{T}$ and, for all $i \neq j$ in $[1, |\mathtt{L}|]$, one has $\mathtt{L}_i \neq \mathtt{L}_j$;*
- *when $\tau = \tau' + \tau''$, then $\mathit{wff}(\tau')$ and $\mathit{wff}(\tau'')$.*

4.2. Rated Types

Since vectors are used to introduce multirate signal processing into Faust, we need to deal with these rate issues in the static semantics. As hinted above, we use rates f in Rate to manage rates:

$$f \in \mathrm{Rate} \quad = \quad \mathbb{Q}^+ \, .$$

In our framework, the only signal processing operations that impact rates are related to over- and sub-sampling conversions. To represent such conversions, we use multiplication and division arithmetic operations, thus defining Rate as the set of positive rational numbers.

The static semantics of signals manipulated in our extended Faust thus not only deals with value types, but also with rates. We link these two concepts in the notion of *rated types* $\tau^\sharp$ in $\mathrm{Type}^\sharp$:

$$\begin{array}{rcl} \tau^\sharp \in \mathrm{Type}^\sharp & = & \mathrm{Type} \times \mathrm{Rate} \mid \\ & & \mathrm{Type}^\sharp \times \mathrm{Type}^\sharp \, . \end{array}$$

We note τ^f the rated type (τ, f) and $\tau^\sharp + \tau'^\sharp$ the addition of two rated types of same rate.

4.3. Impedances

A Faust signal processor maps *beams* of signals, i.e., sets of signals, to beams of signals. These beams have a type (we only represent the type of the image of a signal, since the domain is always time, and signals can only embed values of a single type) called an *impedance* z in Z. Type checking a Faust expression amounts to verifying the compatibility of the input and output impedances of its composed subexpressions:

$$z \in \mathrm{Z} \;=\; \bigcup_{n \geq 0} \mathrm{Type}^{\sharp n} .$$

The null impedance, in $\mathrm{Type}^{\sharp 0}$, is $()$, and is used when no signal is present. A simple impedance is (t^f), and is the type of a beam containing one signal that maps time to values of type t at rate f. The impedance length $|z|$ is defined such that $z \in \mathrm{Type}^{\sharp |z|}$. The i-th rated type in z $(1 \leq i \leq |z|)$ is noted $z[i]$. Two impedances z_1 and z_2 can be concatenated as $z = z_1 \| z_2$, to yield an impedance in $\mathrm{Type}^{\sharp d_1 + d_2}$ where $d_i = |z_i|$, defined as follows:

$$\left\{ \begin{array}{lcl} z[i] & = & z_1[i] \; (1 \leq i \leq d_1) \; , \\ z[i + d_1] & = & z_2[i] \; (1 \leq i \leq d_2) \; . \end{array} \right.$$

To build more complex impedances, we introduce the $\|_n^{n',d}$ iterator as follows:

$$\|_n^{n',d} M = \left\{ \begin{array}{l} (), \text{ if } n > n' \; , \\ M(n) \; \| \; \|_{n+d}^{n',d} M \text{ otherwise} \; . \end{array} \right.$$

where M is any function that maps integers to impedances. Intuitively, $\|_n^{n',d} M$ is the concatenation of $M(n), M(n + d), M(n + 2d), ...M(n')$; when $d = 1$, it can be omitted. As a shorthand, $z[n, n', d]$, which selects from z the types from the n-th type to the n'-th one by step of d , is $\|_n^{n',d} \lambda i.(z[i])$, while a simple slice of z is $z[n, n'] = z[n, n', 1]$. Applying a function M to all elements of an impedance z is noted $\|_z M$, which is a shorthand for $\|_1^{|z|}(M(z[i]))$.

Finally, to simplify our notations, we assume in the following that all the above introduced shorthands can also be used with any term, such as `L` or `T`, member of an iterated product domain.

Definition 2 (Well-Formed Impedance $\mathit{wff}(z)$).
An impedance z is well-formed*, noted $\mathit{wff}(z)$, iff, for all $i \in [1, |z|]$, there exist a rate f_i, noted $\sharp(z[i])$, and a type τ_i such that $z[i] = {\tau_i}^{f_i}$, with $\mathit{wff}(\tau_i)$ and $f_i \in \mathrm{Rate}$.*

Definition 3 (Isochronous Impedance $iso(z)$). *An impedance z is* isochronous*, noted $iso(z)$, iff there exists a rate f, noted $\sharp(z)$, such that, for all $i \in [1, |z|]$, one has $\sharp(z[i]) = f$.*

4.4. Schemes

Some Faust processors, such as the identity processor _ or the delay processor `mem`, are polymorphic. The static definitions of Faust primitives must thus be type schemes that abstract their input and output impedances over abstractable sorts S, in Sort. Type schemes k in Scheme are defined as follows:

$$\begin{aligned} S \in \mathrm{Sort} &= \{\mathrm{Base}, \mathbb{N}, \mathrm{Type}, \mathrm{Rate}, \mathrm{Type}^\sharp\}\ , \\ k \in \mathrm{Scheme} &= (\mathrm{Var} \times \mathrm{Sort})^* \times \mathrm{Z} \times \mathrm{Z}\ . \end{aligned}$$

We note[4] $\Lambda x : S ... x' : S'.(z, z')$ the scheme $(((x, S), ..., (x', S')), z, z')$, where x are abstracting variables in Var. These schemes may be instantiated where needed; the substitution $(z, z')[l'/l]$ of a list l of variables by elements in l' in a pair (z, z') is defined as usual.

The static definitions of Faust primitives are gathered in type environments T that map Faust identifiers to schemes.

5. Static Semantics

The static semantics specifies, by induction on Faust syntax, how impedance pairs are assigned to signal processor expressions. We first define some utilitary operations on static domains, and then provide static rules for Faust.

5.1. Syntax

Faust syntax uses identifiers `I` from the set Ide and expressions `E` in Exp. Numerical constants, be they integers or floating point numbers, are seen as predefined identifiers. The syntax of core Faust is defined in Table 1.

In our core definition of Faust, every expression represents a signal processor, i.e., a function that maps signals, which are functions from time to values, to other signals.

[4]Keeping with a long tradition, we choose the usual ":" sign to denote typing relations, even though it is also used to represent the sequence operation in Faust. The reader should have no problem distinguishing both uses.

$$
\begin{array}{rcl}
\mathtt{E} & ::= & \mathtt{I} \mid \\
 & & [\mathtt{L}> \;\mid\; <\mathtt{L}] \mid \\
 & & \mathtt{E}_1 : \mathtt{E}_2 \mid \mathtt{E}_1, \mathtt{E}_2 \mid \\
 & & \mathtt{E}_1 <: \mathtt{E}_2 \mid \mathtt{E}_1 :> \mathtt{E}_2 \mid \\
 & & \mathtt{E}_1 \sim \mathtt{E}_2 \\
 & & \\
\mathtt{L} \in \mathrm{Ides} & ::= & \bigcup_{n>0} \mathtt{I}^n
\end{array}
$$

Table 1: Faust Syntax

5.2. Impedance Matching

Complex Faust expressions are constructed by connecting together simpler processor expressions. In the case of fan-in (respectively fan-out) expressions, such connections require that the involved signal processors match in some specific sense: Faust uses the *impedance matching* relation $z'_1 \succ z_2$ (resp. $\prec$) to ensure such compatibility conditions. Such a relation goes beyond simple type equality by authorizing a larger (resp. smaller) output z'_1 to fit into a smaller (resp. larger) input z_2, using the following definitions ($\succ$ requires mixing of signals, while $\prec$ simply dispatches the unmodified signals) in which $d'_1 = |z'_1|$ and $d_2 = |z_2|$:

$$
\begin{array}{rcl}
z'_1 \succ z_2 & = & d'_1 d_2 \neq 0 \text{ and} \\
 & & \mathrm{mod}(d'_1, d_2) = 0 \text{ and} \\
 & & \sum_{i\in[0,d'_1/d_2-1]} z'_1[1+id_2,(i+1)d_2] = z_2 \;, \\
z'_1 \prec z_2 & = & d'_1 d_2 \neq 0 \text{ and} \\
 & & \mathrm{mod}(d_2, d'_1) = 0 \text{ and} \\
 & & \|_1^{d_2,d'_1} \lambda i.z'_1 = z_2 \;,
\end{array}
$$

where equality on impedances is defined by structural induction and “mod” denotes the arithmetic modulo operation.

Since we deal in our framework with dependent types (values, via spans, appear in the static domains), performing the mixing of signals, as above, require the ability to perform, in the static semantics, additions over impedances and, consequently, over types; for instance, mixing a signal of type $\mathrm{int}[0,2]$ with one of type $\mathrm{int}[3,6]$ yields a signal of type $\mathrm{int}[3,8]$. To formalize such operations, we assume the existence of static semantics addition rules such as:

(+)

$$[n,m]+[n',m'] \quad = \quad [n+n',m+m'] \quad ,$$

(-)

$$[n,m]-[n',m'] \quad = \quad [n-m',m-n'] \quad ,$$

(/)

$$[n,m]/[n',m'] \quad = \quad [-\omega,+\omega] \qquad (\text{if } [0,0]\subset[n',m'])\ ,$$

(v+)

$$\mathrm{vector}_n(\tau)+\mathrm{vector}_n(\tau') \quad = \quad \mathrm{vector}_n(\tau+\tau') \quad ,$$

(t+)

$$\frac{\tau+\tau' \quad = \quad \tau''}{\tau^f+\tau'^f \quad = \quad \tau''^f}\ ,$$

(z+)

$$\frac{\begin{array}{c}|z|=|z'|=|z''| \\ \forall i\in[1,|z|].z[i]+z'[i]=z''[i]\end{array}}{z+z' \quad = \quad z''}\ .$$

5.3. Subtyping

The presence of values in types logically induces a reflexive, antisymmetric order relationship $\tau \subset \tau'$ on types, rated types and impedances, defined by the rules such as:

(i2f)

$$\mathrm{int}[n,m] \quad \subset \quad \mathrm{float}[n,m] \quad ,$$

(b)

$$\frac{[n,m] \quad \subset \quad [n',m']}{b[n,m] \quad \subset \quad b[n',m']}\ ,$$

(r)

$$\frac{\begin{array}{rcl}\mathtt{L}' & \subset & \mathtt{L} \\ \forall \mathtt{I}'\in\mathtt{L}'.\ \mathtt{T}[\mathtt{L}^{-1}(\mathtt{I}')] & \subset & \mathtt{T}'[\mathtt{L}'^{-1}(\mathtt{I}')]\end{array}}{(\mathtt{L},\mathtt{T}) \quad \subset \quad (\mathtt{L}',\mathtt{T}')}\ .$$

where we introduce[5] the notation $\mathtt{L}^{-1}(\mathtt{I}) = i$ when $\mathtt{L}[i] = \mathtt{I}$. To these rules, we add traditional structural rules on vectors, records and impedances; subtyping is only defined on rated types that have the same rate.

Note that if $\tau \subset \tau'$ and $\tau' \subset \tau$, then $\tau = \tau'$.

5.4. Type Environments

We assume that there is an initial type environment T_0 that provides the typing definitions for the predefined signal processors. For instance, one has:

$$
\begin{aligned}
T_0(\_) &= \Lambda\tau^\sharp : \mathrm{Type}^\sharp.((\tau^\sharp), (\tau^\sharp)) \ , \\
T_0(\mathtt{!}) &= \Lambda\tau^\sharp : \mathrm{Type}^\sharp.((\tau^\sharp), ()) \ , \\
T_0(\mathtt{0}) &= \Lambda f : \mathrm{Rate}.((), (\mathrm{int}[0,0]^f)) \ , \\
T_0(\mathtt{-2.8}) &= \Lambda f : \mathrm{Rate}.((), (\mathrm{float}[-3,-2]^f)) \ , \\
T_0(\mathtt{+}) &= \Lambda\tau : \mathrm{Type}.\tau' : \mathrm{Type}.f : \mathrm{Rate}.((\tau^f, \tau'^f), (\tau^f + \tau'^f)) \ , \\
T_0(\mathtt{mem}) &= \Lambda\tau^\sharp : \mathrm{Type}^\sharp.((\tau^\sharp), (\tau^\sharp)) \ .
\end{aligned}
$$

As a consequence of the implicit mixing introduced by the impedance matching relation $\succ$ used in fan-in operations, signal processors for numerical operators such as + must be able to deal with any type; they are thus associated to polymorphic type schemes in the type environment. Their arguments must also have the same rate, a constraint enforced by the use of the same $t^\sharp$ in these type schemes.

Introducing the vector extension in the static semantics simply amounts to adding, beside the empty vector `{}`, of type $\Lambda f : \mathrm{Rate}.\tau : \mathrm{Type}.((), (\mathrm{vector}_0(\tau)^f))$, four bindings in the initial environment T_0:

- $T_0(\mathtt{vectorize}) =$
  $\Lambda f : \mathrm{Rate}.f' : \mathrm{Rate}.\tau : \mathrm{Type}.n : \mathbb{N}.((\tau^f, \mathrm{int}[n,n]^{f'}), (\mathrm{vector}_n(\tau)^{f/n}))$;

- $T_0(\mathtt{\#}) =$
  $\Lambda f : \mathrm{Rate}.\tau : \mathrm{Type}.m : \mathbb{N}.n : \mathbb{N}.$
  $((\mathrm{vector}_m(\tau)^f, \mathrm{vector}_n(\tau)^f), (\mathrm{vector}_{m+n}(\tau)^f))$;

- $T_0(\mathtt{[]}) =$
  $\Lambda f : \mathrm{Rate}.\tau : \mathrm{Type}.n : \mathbb{N}.((\mathrm{vector}_n(\tau)^f, \mathrm{int}[0, n-1]^f), (\tau^f))$;

[5] Note that $\mathtt{L}^{-1}$ is here well defined since we assume from the start that there are no duplicates in field names.

- $T_0(\texttt{serialize}) = \Lambda f : \text{Rate}.\tau : \text{Type}.n : \mathbb{N}.((\text{vector}_n(\tau)^f), (\tau^{f \times n}))$.

The dependent type system is key here. In the primitive `vectorize`, we are able to specify that the vector size has to be constant, since its type uses a span restricted to be one-valued, $[n, n]$; note that the rate f' of this signal is also irrelevant, and can be of any value. When concatenating vectors with the `#` processor, the resulting vector size $m + n$ sums the sizes of the input vectors. We are also able to ensure that no out-of-bound accesses can occur in Faust, since the index signal argument fed to the `[]` signal processor is constrained, at compile time, to be between 0 and the vector size, since its span is $[0, n - 1]$. Finally, notice how size information impacts signal rates; this is key to prove the theorem of Section 8.

5.5. Typing Rules

Faust is strongly and statically typed. Every expression, a signal processor, is typed by its I/O impedances:

Definition 4 (Expression Type Correctness $T \vdash$ `E`).
An expression `E` *is* type correct *in an environment T, noted $T \vdash$ `E`, if there exist impedances z and z' such that $T \vdash$ `E` $: (z, z')$ with wff(z) and wff(z').*

The static semantics inference rules are defined in Table 2; some are rather straightforward. Rule (i) ensures that identifiers are typable in the type environment T; type schemes can be instantiated to adapt themselves to a given typing context of Identifier `I`. The typical rule ($\subset$) allows types to be extended according to the order relationship induced by spans in types, records and basic types. In Rule (:), signal processors are plugged in sequence, which requires that the output impedance of $\texttt{E}_1$ is the same as $\texttt{E}_2$'s input. In Rule (,), running two signal processors in parallel requires that their input and output impedances are concatenated. In Rules (<:) and (:>), the $\prec$ and $\succ$ constraints are used to ensure that a proper matching of the output of $\texttt{E}_1$ to the input of $\texttt{E}_2$ is possible.

In Rules ([>) and (<]), we deal with records. First, we specify how the [> construct builds a single signal u of records with the proper field names `L` from an isochronous beam of signals of the same length. With (<]), we perform the dual operation, generating a beam of isochronous signals from a (subset $\texttt{L}'$ of field names in a) single signal u of records.

The most involved rule deals with loops ($\sim$). Here, the input impedance z_2 of the feedback expression $\texttt{E}_2$ is constrained to be the first $|z_2|$ types of the output impedance z'. Also, the first $|z'_2|$ elements of the input impedance of the main expression $\mathtt{E}_1$ must be the same as the ouput impedance of the feedback expression $\mathtt{E}_2$; these looped-back signals will not thus impact the global input impedance $z_1[|z'_2|+1, |z_1|]$. To simplify the dynamic semantics (see Table 3), all looped-back signals are required to have the same rate. We do not expect this constraint to be a problem in practice since all operations have eventually to be performed in such a manner; any rate mismatch at the "$\sim$" level can be fixed by moving there the down/up sampling conversions that would have to be present anyhow in $\mathtt{E}_2$. Finally, note that the output impedance $\widehat{z'}$ is here an approximation of z'. This is introduced not for semantic reasons, but to make type checking decidable while ensuring that the dependent return type is valid independantly of the unknown bounds of the iteration space:

Definition 5 (Impedance Widening $\widehat{z}$).
The widened impedance *of Impedance z, noted $\widehat{z}$, is such that $|\widehat{z}| = |z|$ and $\forall i \in [1, |z|].\widehat{z}[i] = \widehat{z[i]}$, with:*

- $\widehat{\mathrm{vector}_n(\tau)}^f = \mathrm{vector}_n(\widehat{\tau})^f$;
- $\widehat{(\mathtt{L}, \mathtt{T})} = (\mathtt{L}, \|_{\mathtt{T}} \lambda\tau.\widehat{\tau})$;
- $\widehat{b[a]}^f = b[\widehat{a}]^f$;
- $\widehat{[n, m]} = [-\omega, +\omega]$.

Basically, all knowledge on value bounds is lost under widening.

6. Dynamic Semantics

Since Faust sees parallelism as an implementation issue (Faust expressions can be trivially evaluated in parallel, since no side-effets can be performed in the core language), the denotational semantics for core Faust is based on standard notions and does not introduce parallel-specific concepts such as powerdomains, while remaining synchronous.

6.1. Domains

A Faust expression denotes a signal processor; as such its semantics manipulates signals, which assign various values to time ticks. The dynamic semantics, in particular, uses integers n, k, d, i (in $\mathbb{N}$) and times t in $\mathrm{Time} = \mathbb{N}$.

Signals map times to values v in Val :

$$\begin{aligned} v \in \mathrm{Val} &= N + R + (\mathbb{N} \rightarrow \mathrm{Val}) + (\mathrm{Ide} \rightarrow \mathrm{Val}) \ , \\ N &= \mathbb{N} + \{\perp\} + \{?\} \ , \\ R &= \mathbb{R} + \{\perp\} + \{?\} \ . \end{aligned}$$

Since evaluation processes may be non-terminating, we posit that sets such as Val are complete partial orders (cpos), with order relation $\sqsubset$ and bottom $\perp$. Since all functional cpos are here strict, we define, for $f \in A \rightarrow B$, its domain $dom(f) = \{a/f(a) \neq \perp\}$; the size of this domain $|dom(f)|$, called the support $\underline{f}$ of f, is a member of $\mathbb{N} + \{\omega\}$, where ω is used to deal with infinite cpos.

The value '?' denotes error values (useful to denote non-existing values such as 1/0), and thus, for any Operator o and Value v different from $\perp$, we assume $o(?, v) = ?$. The functional cpo $\mathbb{N} \rightarrow \mathrm{Val}$ is used for vector denotations, while $\mathrm{Ide} \rightarrow \mathrm{Val}$ is used for records.

A signal s, which is intuitively a "history" denoted by a function, is a member of $\mathrm{Signal} = \mathrm{Time} \rightarrow \mathrm{Val}$. We define the domain $dom(s)$ and support $\underline{s}$ of a signal s as above.

Signal is a cpo ordered by:

$$\begin{aligned} s \sqsubset s' &= dom(s) \subset dom(s') \text{ and} \\ & \quad \forall t \in [0, \underline{s} - 1],\ s(t) = s'(t) \ . \end{aligned}$$

We gather signals into beams $m = (m_1, ..., m_n)$ in Beam:

$$m \in \mathrm{Beam} = \bigcup_{n \geq 0} \mathrm{Signal}^n \ .$$

We consider that all notations introduced to manipulate impedances can similarly be applied to beams. Note that we do not need to consider Beam as a cpo, although each Signal^n is, with the order:

$$\begin{aligned} m \sqsubset m' &= \forall i \in [1, n],\ m[i] \sqsubset m'[i] \ , \\ \perp &= (\lambda t.\ \perp, ..., \lambda t.\ \perp) \in \mathrm{Signal}^n \ . \end{aligned}$$

A signal processor p in Proc is the basic constituent of Faust programs: $p \in \mathrm{Proc} = \mathrm{Beam} \rightarrow \mathrm{Beam}$. We define $dim(p) = (n, n')$ such that $p \in \mathrm{Signal}^n \rightarrow \mathrm{Signal}^{n'}$.

The standard semantics of a Faust expression is a function of the semantics of its free identifiers; we collect these in a state r, a member of $\mathrm{State} = \mathrm{Ide} \rightarrow \mathrm{Proc}$.

6.2. Denotational Rules

We assume given an initial state r_0, which binds Faust predefined identifiers to their value such that, for instance:

$$
\begin{array}{rcl}
r_0(\_) & = & \lambda(s).(s)\ , \\
r_0(\texttt{!}) & = & \lambda(s).()\ , \\
r_0(\texttt{0}) & = & \lambda().(\lambda t.0)\ , \\
r_0(\texttt{+}) & = & \lambda(s_1, s_2).(\lambda t.s_1(t) + s_2(t))\ , \\
r_0(\texttt{/}) & = & \lambda(s_1, s_2). \\
& & \quad(\lambda t. \\
& & \quad\ s_1(t)/s_2(t) \text{ if } t < \min(\underline{s_1}, \underline{s_2}) \text{ and } s_2(t) \neq 0, \\
& & \quad\ ? \text{ if } t < \min(\underline{s_1}, \underline{s_2}), \\
& & \quad\ \perp \text{ otherwise})\ , \\
r_0(\texttt{mem}) & = & \lambda(s_1).(\text{delay}(s_1, \lambda t.1))\ .
\end{array}
$$

These definitions assume that $T \vdash \texttt{0} : t^f$ for all types t and rates f, since this is needed for the definition of delayed signals in "$\sim$" loops to make sense. Similarly + is supposed to be defined for all types since it is used in the dynamic definition of :> (see below). We assume the existence of the delay function defined as:

$$
\text{delay}(s_1, s_2) \;=\; \lambda t. \perp \text{ if } s_2(t) = \perp, s_1(t - s_2(t)) \text{ if } t - s_2(t) \geq 0, 0 \text{ otherwise}\ ,
$$

which delays each sample of Signal s_1 by a number of time slots given, at each time t, by $s_2(t)$; the usual one-slot delay used in the semantics of "$\sim$" loops is thus $\text{delay}(s_1, \lambda t.1)$, as is the semantics of `mem`.

As in the static semantics, introducing the vector extension in the dynamic semantics[6] simply amounts to adding, beside the value $\lambda().(\lambda t.())$ for `{}`, four straightforward bindings in the initial state:

- $r_0(\texttt{vectorize}) =$
  $\lambda(s_1, s_2).(\lambda t.(s_1(nt), ..., s_1(n - 1 + nt)),\ \text{where } n = s_2(0))$;
- $r_0(\texttt{\#}) = \lambda(s_1, s_2).(\lambda t.s_1(t) \| s_2(t))$;
- $r_0(\texttt{[]}) = \lambda(s_1, s_2).(\lambda t.s_1(t)[s_2(t)])$;

[6]We consider that all notations introduced to manipulate impedances can similarly be applied to vectors.

- $r_0(\texttt{serialize}) =$
  $\lambda(s).(\lambda t.\ \bot,\ \text{if } n = |s(0)| = 0,\ s(\lfloor t/n \rfloor)[\text{mod}(t, n)] \text{ otherwise}).$

To be able to properly define the semantic function E:

$$E \in \text{Exp} \rightarrow \text{State} \rightarrow \text{Beam} \rightarrow \text{Beam}\ ,$$

one needs to ensure that we operate with states that are type-correct.

Definition 6 (State Type Correctness $T \vdash r$).
A state r is type correct *in an environment T, noted $T \vdash r$, if, for all* `I` *in $dom(r)$, one has $T \vdash$* `I`.

The semantics $E[\![\texttt{E}]\!]r$ of an expression `E` in a type-correct state r is a function that maps an input beam m to an output beam m' (see Table 3).

The semantics of an identifier is available in the state r. The semantics of ":" is the usual composition of the subexpressions' semantics. The semantics of a parallel composition is a function that takes a beam of size $d_1 + d_2$ and feeds the first d_1 signals into p_1 and the subsequent d_2 into p_2; the outputs are concatenated. The fan-out construct repeatedly concatenates the outputs of p_1 to feed into the (larger) d_2 inputs of p_2. The fan-in construct performs a kind of opposite operation; all $\text{mod}(i, d_2)$-th output values of p_1 are summed together to construct the i-th input value of p_2. The management of records is rather straightforward. Note though that, when building records with $[>$, one needs to enforce that all incoming signals have a defined value at time t to build a proper, strict record. The loop expression has the most complex semantics. Its feedback behavior is represented by a fix point construct; the output of p_2 is fed to p_1, after being concatenated to m, to yield m'; the input of p_2 is the one-slot delayed appropriate part of m', made of signals the static semantics ensures are isochronous.

6.3. Properties

An interesting corollary of Faust denotational semantics is that one can easily prove that the ":" constructor is actually not necessary:

Theorem 1 (: as Syntactic Sugar). *Assume $T \vdash \texttt{E}_1 : \texttt{E}_2 : (z, z')$. Then $T \vdash \texttt{E}_1 <: \texttt{E}_2 : (z, z')$ and $T \vdash \texttt{E}_1 :> \texttt{E}_2 : (z, z')$. Moreover, $E[\![\texttt{E}_1 : \texttt{E}_2]\!] = E[\![\texttt{E}_1 <: \texttt{E}_2]\!] = E[\![\texttt{E}_1 :> \texttt{E}_2]\!]$.*

Also, as can be seen in the denotational semantics of Faust, only total functions can be expressed in the language; there is no general, potentially-infinite looping mechanism in this framework. This is, of course, not an oversight, but a conscient design decision that enables programs to be implemented in a more efficient way, while not being too strong a limitation, given the audio application domain we target. We introduce below a formal definition for such a totality property.

Definition 7 (Defined Beam *def*(m)**).** *A beam m is* defined*, noted def(m), iff there exist no i in $[1, |m|]$ such that $m[i] = \perp$.*

Theorem 2 (Totality). *If def(m), then def$(E[\![\mathtt{E}]\!]rm)$.*

Proof. By induction on the structure of $\mathtt{E}$. The $\sim$ construct can be handled by noticing that $(\mathrm{delay}(s, \lambda t.1))$ is always defined, even if (s) is not. □

7. Subject Reduction Theorem

One needs to ensure the consistency of both static and dynamic semantics along the evaluation process; this amounts to showing that the types of values, signals and beams are preserved.

Definition 8 (Value Type Correctness $v : \tau$**).** *A value v is* type correct*, noted $v : \tau$, iff:*

- *when $v \in \mathbb{N}$, then $\tau = \mathrm{int}[n, m]$ and $n \leq v \leq m$;*
- *when $v \in \mathbb{R}$, then $\tau = \mathrm{float}[n, m]$ and $n \leq v \leq m$;*
- *when $v \in \mathbb{N} \to \mathrm{Val}$, then $\tau = \mathrm{vector}_n(\tau'), dom(v) = [0, n-1]$ and, for all $i \in [0, n-1], v(i) : \tau'$;*
- *when $v \in \mathrm{Ide} \to \mathrm{Val}$, then $\tau = (\mathtt{L}, \mathtt{T}), dom(v) = \mathtt{L}$ and, for all $\mathtt{I} \in \mathtt{L}, v(\mathtt{I}) : \mathtt{T}[\mathtt{L}^{-1}(\mathtt{I})]$.*

Definition 9 (Signal Type Correctness $s : \tau^f$**).** *A signal s is* type correct *w.r.t. a type τ^f, noted $s : \tau^f$, if, for all $u \in dom(s)$, one has $s(u) : \tau$.*

Definition 10 (Beam Type Correctness $m : z$**).** *A beam m is* type correct *w.r.t. an impedance z, noted $m : z$, if $|m| = |z|$ and, for all $i \in [1, |m|]$, one has $m[i] : z[i]$.*

For the evaluation process to preserve consistency, the environment T and state r, which provide the static and semantic values of predefined identifiers, must provide consistent definitions for their domains. We use the following definition to ensure this constraint:

Definition 11 (State Type Consistency $\vdash T, r$). *An environment T and a state r are* consistent*, noted $\vdash T, r$, if, for all $\mathtt{I}$ in $dom(r)$, for all z, z', m, one has: if $T \vdash \mathtt{I} : (z, z')$ and $m : z$, then $r(\mathtt{I})(m) : z'$ and $dim(r(\mathtt{I})) = (|z|, |z'|)$.*

We are now equiped to state our first typing theorem. The Subject Reduction theorem basically states that, given a Faust expression $\mathtt{E}$, if the environment T and state r are consistent and $\mathtt{E}$ maps beams of impedance z to beams of impedance z', then, given a beam m that is type correct w.r.t. z, then the semantics $p(m)$ of $\mathtt{E}$ will yield a beam m' of impedance z':

Theorem 3 (Subject Reduction). *For all $\mathtt{E}, T, z, z', r$ and m, if*

$$\begin{array}{l} \vdash T, r \ , \\ m : z \ and \\ T \vdash \mathtt{E} : (z, z') \ , \end{array}$$

then $p(m) : z'$ and $dim(p) = (|z|, |z'|)$, where $p = E[\![\mathtt{E}]\!]r$.

Proof. By induction on the structure of $\mathtt{E}$.

- $\mathtt{E} = \mathtt{I}$. Use $E[\![\mathtt{I}]\!]r = r(\mathtt{I})$ and State Type Consistency.
- $\mathtt{E} = \mathtt{E}_1 : \mathtt{E}_2$. $T \vdash \mathtt{E} : (z, z')$ implies, using Rule (:), there exists z'_1 such that $T \vdash \mathtt{E}_1 : (z, z'_1)$ and $T \vdash \mathtt{E}_2 : (z'_1, z')$.

  By induction on $\mathtt{E}_1$, $p_1(m) : z'_1$ and $dim(p_1) = (|z|, |z'_1|)$.

  By induction on $\mathtt{E}_2$, one gets $p_2(p_1(m)) : z'$ and $dim(p_2) = (|z'_1|, |z'|)$.

  The definition of $E[\![\mathtt{E}]\!]$ yields $E[\![\mathtt{E}]\!]rm : z'$ and

  $$dim(p) = (|m|, |p_2(p_1(m))|) = (|z|, |z'|) \ .$$
- $\mathtt{E} = \mathtt{E}_1, \mathtt{E}_2$. $T \vdash \mathtt{E} : (z, z')$ implies, using Rule (,), there exist z_1, z_2, z'_1, z'_2 such that $z = z_1 \| z_2, z' = z'_1 \| z'_2, T \vdash \mathtt{E}_1 : (z_1, z'_1)$ and $T \vdash \mathtt{E}_2 : (z_2, z'_2)$.

By Beam Type Correctness on $m : z$, one gets $|m| = |z|$ and, for all i in $[1, |m|]$, $m[i] \in \text{Time} \to z[i]$.

By definition of z, $|z| = |z_1| + |z_2|$. Using the first $|z_1|$ elements of z, one gets $m[1, |z_1|] : z_1$. By induction on $\texttt{E}_1$, one gets $p_1(m[1, |z_1|]) : z'_1$ and $dim(p_1) = (|z_1|, |z'_1|)$. Since, in the definition of E, $d_1 = |z_1|$, thus $p_1(m[1, d_1]) : z'_1$.

Using the subsequent $|z_2|$ elements of z, one gets $m[d_1+1, d_1+|z_2|] : z_2$. By induction on $\texttt{E}_2$, $E[\![\texttt{E}_2]\!]r(m[d_1+1, d_1+|z_2|]) : z'_2$ and $dim(p_2) = (|z_2|, |z'_2|)$. Since $d_2 = |z_2|$, then $E[\![\texttt{E}_2]\!]r(m[d_1 + 1, d_1 + d_2]) : z'_2$.

The definition of E on $\texttt{E}$ yields $p(m) = p_1(m[1, d_1]) \| p_2(m[d_1+1, d_1+d_2])$. By definition of Beam Type Correctness, $p(m) : z'_1 \| z'_2 = z'$ and $dim(p) = (|m|, |z'|) = (|z|, |z'|)$.

- $\texttt{E} = \texttt{E}_1 <: \texttt{E}_2$. $T \vdash \texttt{E} : (z, z')$ implies, using Rule (<:), there exist z'_1, z_2, k such that $T \vdash \texttt{E}_1 : (z, z'_1)$, $T \vdash \texttt{E}_2 : (z_2, z')$, $|z_2| = k|z'_1|$ and, for all i in $[0, k-1]$, $z_2[1 + i|z'_1|, |z'_1| + i|z'_1|] = z'_1$.

  By induction on $\texttt{E}_1$, one gets $p_1(m) : z'_1$ and $dim(p_1) = (|z|, |z'_1|)$. By induction on $\texttt{E}_2$, $dim(p_2) = (|z_2|, |z'|)$. By definition of E, $d'_1 = |z'_1|$ and $d_2 = |z_2|$; thus $d_2 = kd'_1$.

  Let $m' = \|_1^{d_2, d'_1} \lambda i.p_1(m) = p_1(m) \| .... \| p_1(m) \in \text{Signal}^{kd'_1}$. By definition of Beam Type Correctness and k, one gets $m' : z_2$. By induction on $\texttt{E}_2$, one gets $p_2(m') : z'$ and $dim(p_2) = (|z_2|, |z'|)$.

  By definition of E on $\texttt{E}$, then $p(m) : z'$ and $dim(p) = (|z|, |z'|)$.

- $\texttt{E} = \texttt{E}_1 :> \texttt{E}_2$. $T \vdash \texttt{E} : (z, z')$ implies, using Rule (:>), there exist z'_1, z_2, k such that $T \vdash \texttt{E}_1 : (z, z'_1)$, $T \vdash \texttt{E}_2 : (z_2, z')$, $|z'_1| = k|z_2|$ and, for all i in $[0, k-1]$, $z'_1[1 + i|z_2|, |z_2| + i|z_2|] = z_2$.

  By induction on $\texttt{E}_1$, one gets $p_1(m) : z'_1$ and $dim(p_1) = (|z|, |z'_1|)$. By induction on $\texttt{E}_2$, $dim(p_2) = (|z_2|, |z'|)$. By definition of E, $d'_1 = |z'_1|$ and $d_2 = |z_2|$; thus $d'_1 = kd_2$.

  For all i in $[1, d_2]$, let $m^i = p_1(m)[i, d'_1, d_2]$. Thus:

$$\begin{array}{rcl} m^i & = & \|_i^{d'_1, d_2} \lambda j.(p_1(m)[j]) \\ & : & \|_i^{d'_1, d_2} \lambda j.(z'_1[j]), \text{ by induction on } \texttt{E}_1 \\ & = & (z'_1[i]) \| (z'_1[i + d_2]) \| ... \| (z'_1[i + (k-1)d_2]), \text{ by definition of } k. \end{array}$$

Thus, by definition of mix and the application of $+$ on impedances, one gets:

$$\begin{aligned} mix(m^i) &: \ (\sum_{l\in[0,k-1]} z'_1[i+ld_2]\ ) \\ &= \ (z_2[i]),\ \text{since } z'_1 \succ z_2\ . \end{aligned}$$

Let $m_2 = \|_1^{d_2} \lambda i.mix(m^i)$. Then $m_2 : (z_2[1], ..., z_2[d_2]) = z_2$.

By induction on $\mathtt{E}_2$, then $p(m) = p_2(m_2) : z'$ and $dim(p) = (|z|, |z'|)$.

- $\mathtt{E} = [\mathtt{L}>$. $T \vdash [\mathtt{L}>: (z, z')$ implies $iso(z)$, $|z| = |\mathtt{L}|$ and there exists $u = (\mathtt{L}, \|_z type)$ such that $z' = (u^{\sharp(z)})$. By definition of the dynamic semantics, one has $p(m) = (s')$ with

$$s' = \lambda t' \in \bigcap_{i\in[1,|m|]} dom(m[i]).\lambda \mathtt{I}' \in \mathtt{L}.m[\mathtt{L}^{-1}(\mathtt{I}')](t')\ .$$

Thus, one sees $dim(p) = (|m|, 1) = (|z|, |z'|)$, since, by hypothesis, $m : z$.

To prove $p(m) : z'$, one needs to show that, for all t' in $dom(s')$, one has $s'(t') : u$. Thus, since $v = s'(t') = \lambda \mathtt{I}'.m[\mathtt{L}^{-1}(\mathtt{I}')](t')$, when $\mathtt{I}' \in \mathtt{L}$, and $\perp$ otherwise, one needs to show, by Value Type Correctness, that there exist $\mathtt{L}_u$ and $\mathtt{T}_u$ such that

$$u = (\mathtt{L}_u, \mathtt{T}_u)\ \wedge\ dom(v) = \mathtt{L}_u\ \wedge\ \forall \mathtt{I}_u \in \mathtt{L}_u.v(\mathtt{I}_u) : \mathtt{T}_u[{\mathtt{L}_u}^{-1}(\mathtt{I}_u)]\ .$$

Choosing $\mathtt{L}_u = \mathtt{L}$ and $\mathtt{T}_u = \|_z type$, the first two propositions are satisfied. To prove $p(m) = z'$, one is left to show, given the definition of v and Value Type Correctness, that for all $\mathtt{I}_u$ in $\mathtt{L}$:

$$m[\mathtt{L}^{-1}(\mathtt{I}_u)](t') : \|_z type\ [\mathtt{L}^{-1}(\mathtt{I}_u)]\ .$$

Since, by definition of $\mathtt{L}^{-1}$, one has $\mathtt{L}^{-1}(\mathtt{I}_u) = i_u$ if and only if $\mathtt{L}[i_u] = \mathtt{I}_u$, we see, collecting quantifiers, that proving $p(m) : z'$ is equivalent to showing that:

$$\forall t' \in dom(s').\forall i_u \in [1, |\mathtt{L}|].m[i_u](t') : type(z[i_u])$$

is true. Yet, by hypothesis, $m : z$, and thus $\forall i \in [1, |z|]$ and $\forall t \in dom(m[i])$, $m[i](t) : type(z[i])$ is true, and implies what is needed. Indeed, first, $|z| = |\mathtt{L}|$ and, second, since $dom(s') = \bigcap_{i\in[1,|m|]} dom(m[i])$, the proposition with the two universal quantifiers exchanged is also true.

- $\mathtt{E} = \mathtt{<L'>}$. $T \vdash \mathtt{<L'>} : (z, z')$ implies that there exist $\mathtt{L}, \mathtt{T}, f$ and u such that $u = (\mathtt{L}, \mathtt{T})$, $z' = \|_{\mathtt{L}'} \lambda \mathtt{I}'.\mathtt{T}[\mathtt{L}^{-1}(\mathtt{I}')]^f$, $\mathtt{L}' \subset \mathtt{L}$ and $z = (u^f)$. By definition of the dynamic semantics, one has $p(m) = m'$ with

  $$m' = \lambda(s).\|_{\mathtt{L}'} \lambda \mathtt{I}'.\lambda t.s(t)(\mathtt{I}') \ ,$$

  where $m = (s)$.

  Thus, one sees $dim(p) = (1, |\mathtt{L}'|) = (|z|, |z'|)$, by definition of z and z'.

  To prove $p(m) : z'$, one needs to show that

  $$\|_{\mathtt{L}'} \lambda \mathtt{I}'.\lambda t.s(t)(\mathtt{I}') : \|_{\mathtt{L}'} \lambda \mathtt{I}'.\mathtt{T}[\mathtt{L}^{-1}(\mathtt{I}')] \ ,$$

  which yields, by definition of Value Type Correctness, that for all $\mathtt{I} \in \mathtt{L}'$, one has:

  $$\lambda t.s(t)(\mathtt{I}) : \mathtt{T}[\mathtt{L}^{-1}(\mathtt{I})] \ .$$

  Thus, one needs to show that, for all $t \in dom(s)$ and $\mathtt{I} \in \mathtt{L}'$, one has $s(t)(\mathtt{I}) : \mathtt{T}[\mathtt{L}^{-1}(\mathtt{I})]$. Yet, by definition of the $m : z$ hypothesis, i.e., $(s) : (u^f)$, one knows that $\forall t \in dom(s).s(t) : u$. By Value Type Correctness on records, one gets that $\forall \mathtt{I} \in \mathtt{L}.s(t)(\mathtt{I}) : \mathtt{T}[\mathtt{L}^{-1}(\mathtt{I})]$, which implies what is needed to prove $p(m) : z'$, since $\mathtt{L}' \subset \mathtt{L}$.

- $\mathtt{E} = \mathtt{E}_1 \sim \mathtt{E}_2$. $T \vdash \mathtt{E} : (z, \widehat{z'})$ implies, using Rule ($\sim$), there exist z_1, z_2, s'_2 such that $T \vdash \mathtt{E}_1 : (z_1, z')$, $T \vdash \mathtt{E}_2 : (z_2, z'_2)$, $z_2 = z'[1, |z_2|]$, $z'_2 = z_1[1, |z'_2|]$ and $z = z_1[|z'_2| + 1, |z_1|]$. One sees that $z_1 = z'_2 \| z$.

  Let $m' = fix(F)$, with $F = \lambda m'.p_1(p_2(@(m'[1, d_2])) \| m)$. We are going to prove $fix(F) : z'$ and $dim(\lambda m.fix(F)) = (|z|, |z'|)$. Using fix point induction (which is valid since we stay in the cpo $\mathrm{Signal}^{|z'|}$), this needs to be proven for the bottom element and, assuming this is true for m', show it is true for $F(m')$.

  - Let $\perp'$ be bottom in $\mathrm{Signal}^{|z'|}$: $\perp' = (\lambda t. \perp, ..., \lambda t. \perp) : z'$. One immediately gets $dim(\lambda m. \perp') = (|m|, |z'|) = (|z|, |z'|)$.
  - Assume $m' : z'$. We need to show that $F(m') : z'$ and $dim(\lambda m.F(m')) = (|z|, |z'|)$. One has $F(m') = p_1(p_2(@(m'[1, d_2])) \| m)$.

    Using the lemma (left to the reader) that, if $m' : z'$, then $@(m') : z'$, we get that $@(m'[1, d_2]) : z_2$.

    By induction on $\mathtt{E}_2$, $F(m') = p_1(m'' \| m)$, where $m'' : z'_2$.

Since $m : z$, then, by induction on $\texttt{E}_1$, one has $F(m') : z'$ and $dim(\lambda m.F(m')) = (|m|, |z'|) = (|z|, |z'|)$.

- By fix point induction then, $m' : z'$. Since one easily sees that $z' \subset \widehat{z'}$, then, using Rule ($\subset$) and $dim(\lambda m.m') = (|z|, |z'|) = (|z|, |\widehat{z'}|)$, one gets the required result. □

The Subject Reduction theorem can be readily applied to typing Faust expressions in the initial environment T_0 and state r_0, since one can easily prove the following theorem:

Theorem 4 (Initial State Type Consistency).

$$\vdash T_0, r_0 \ .$$

8. Rate Correctness Theorem

In the presence of signals that use different rates at run time, the consistency of their rate assignment must be ensured. In particular, we show below that the support of signals and, more generally, beams can be bounded in a way consistent with their relative rates; this is the Rate Correctness theorem.

8.1. Beam Boudness Definition

Even though Faust expressions only denote total signal processors, the semantics of "$\sim$" loops is defined as a fixed point, which is based on partially defined signals. This leads us to the notion of beam boundness.

Definition 12 (Beam Bound $\mu(m, z)$).
The bound $\mu(m, z)$ of a beam m of impedance z is $\min_{i\in[1,|z|]} \lfloor \underline{m[i]}/\sharp(z[i])\rfloor$, where, for all $n \in \mathbb{N}$, we have $n/0 = \omega$.

Informally, when $\mu(m, z) = c$, then there is at least one signal i^* in m that has at most $(c+1)\sharp(z[i^*])-1$ elements in its domain of definition[7]. This is interesting since the supports of signals in a beam m tell us something about how many values can be computed if we use m as input of a signal processor. Thus $c\sharp(z[i^*])$ is an upper bound on the number of elements that can be used in a synchronous

[7] When signals are properly synchronized, e.g., in an actual computation, all $\underline{m[i]}/\sharp(z[i])$ are equal, and the comments in this section about i^* apply in fact to all signals.

computation (all subsequent values are $\perp$), thus yielding some clues about the size of buffers needed to perform it.

Another way to look at c-boundness comes from c itself; being the inverse of a rate, its unit is the second, and thus c is a time. The definition of beam boundness yields an upper bound on the number of time ticks required to exhaust at least one of the signals of m, thus providing a (logical) time limit on computations that would use these as actual inputs. Even though this limit, as stated here, holds for a complete computation, it also applies when one deals with slices of the computation process, for instance when considering buffered versions of a program.

We illustrate this notion of beam boundness in Figure 2, where incoming signals s_i have different rates. The suport of s_1 is 5, while s_2's support is 3. Note that at most 4 elements are available in the ouput signal s'_1, since s_1 would need one additional element for the computation of two additional elements in s'_1 to be valid.

Of course, in general, explicit delaying operations introduced via the 1-sample delay `mem` primitive may occur in Faust programs. Since these operations cumulatively extend the support of signals, we need to provide an upper-bound estimate of such an extension, as a count of the additional elements introduced by a given signal processor `E`; the number of such elements is, of course, related to the rate of each given `mem` use. We define then $@_T(\mathtt{E})$ as follows:

$$@_T(\mathtt{E}) = \sum_{\{z/\mathtt{mem}:(z,z)\ \in ids(T,\mathtt{E})\}} \lceil 1/\sharp(z) \rceil\ .$$

Here, $ids(T, \mathtt{E})$ denotes the list of Faust identifier typings $\mathtt{I} : (z_i, z'_i)$ obtained[8] via the application of Rule (i) during the typing derivation $T \vdash \mathtt{E} : (z, z')$. To get a correct upper-bound, $@_T(\mathtt{E})$ simply sums the impact of each `mem` operation, which is a safe albeit not very tight upper-bound. Note that $@_T(\mathtt{E})$ is a syntactic notion, defined by induction on `E` and is independant of the size of input signals.

8.2. Theorem

The Rate Correctness theorem states that, given a Faust expression `E`, if the environment T and state r are consistent and `E` maps beams of impedance z to beams of impedance z', then, given a beam m that is type correct w.r.t. z and is

[8]We leave to the reader the exact specification of this function, which extends Faust static semantics with simple bookkeeping operations, e.g., via rules such as $T \vdash \mathtt{I} : ((z, z'), \{\mathtt{I} : (z, z')\}$.

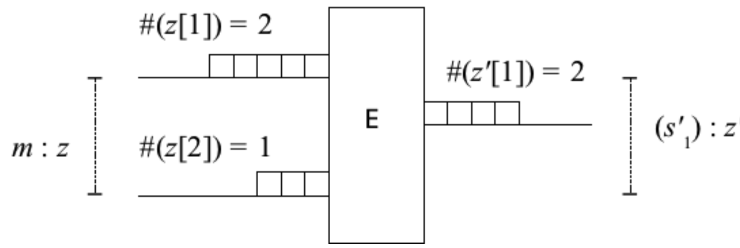

Figure 2: A beam bound example, with $m = (s_1, s_2)$ and $\mu(m, z) = 2$

bounded, then the semantics $p(m)$ of E will yield a similarly bounded beam m' of impedance z'.

Theorem 5 (Rate Correctness).
For all $\mathtt{E}, T, z, z', c, r, m$ *and* m'*, if*

$$\vdash T, r \ ,$$
$$m : z \ ,$$
$$T \vdash \mathtt{E} : (z, z') \ ,$$

then $|z'| = 0 \ \vee \ \mu(m', z') \leq \mu(m, z) + @_T(\mathtt{E})$*, where* $m' = p(m) : z'$ *and* $p = E[\![\mathtt{E}]\!]r$*.*

Basically, this theorem tells us that the running time of E is always upper-bounded, whichever way we try to assess it via any of its observable facets (namely input or output data), modulo the presence of explicit delays: $\mu(m, z)$ is consistent and thus a characteristics of E. This shows that the synchronous nature of Faust beams is preserved by evaluation.

Proof. By induction on the structure of E.

- $\mathtt{E} = \mathtt{I}$. Use $E[\![\mathtt{I}]\!]r = r(\mathtt{I})$ and then:
  - trivial for _;
  - for constants (thus with $z = ()$), since the minimum of the empty set is ω, then $\mu(m, z) = \omega$. The property $\mu(m', z') \leq \omega$ is always satisfied;
  - for `!`, since $|z'| = 0$, then the theorem is trivially satisfied;
  - for `mem`, one has $\mu(m', z') = \min_{i \in [1, |z'|]} \lfloor m'[i] / \sharp(z'[i]) \rfloor$. Since $z' = z$, $|z| = 1$ and $m'[1] = m[1] + 1$ by definition of `mem`, one gets $\lfloor m'[1] / \sharp(z'[1]) \rfloor = \lfloor (m[1]+1) / \sharp(z[1]) \rfloor \leq \lfloor m[1] / \sharp(z[1]) \rfloor + \lceil 1 / \sharp(z[1]) \rceil$. Thus, $\mu(m', z') \leq \mu(m, z) + @_T(\mathtt{I})$, as required;

- for synchronous operations such as `+` or `[]`, this is obvious since $\sharp(z[i]) = \sharp(z'[i'])$.
- for vectorizing and serializing operations, the relationship on rates is, by design, the exact inverse of the one on the size of the domains, thus yielding in fact the stronger relation $\mu(m', z') = \mu(m, z)$.

- $\mathtt{E} = \mathtt{E}_1 : \mathtt{E}_2$. $T \vdash \mathtt{E} : (z, z')$ implies, using Rule (:), there exists z'_1 such that $T \vdash \mathtt{E}_1 : (z, z'_1)$ and $T \vdash \mathtt{E}_2 : (z'_1, z')$.

  By induction on $\mathtt{E}_1$, we get $m'_1 = p_1(m) : z'_1$ and $|z'_1| = 0 \ \vee \ \mu(m'_1, z'_1) \leq \mu(m, z) + @_T(\mathtt{E}_1)$.

  If $|z'_1| = 0$, the proof for $\mathtt{E}_2$ follows the lines of the one we used above for constants.

  Otherwise, by induction on $\mathtt{E}_2$, one gets $m' = p_2(m'_1) : z'$ and $|z'| = 0 \ \vee \ \mu(m', z') \leq \mu(m'_1, z'_1) + @_T(\mathtt{E}_2)$. Since $@_T(\mathtt{E}) = @_T(\mathtt{E}_1) + @_T(\mathtt{E}_2)$, one gets $\mu(m', z') \leq \mu(m'_1, z'_1) + @_T(\mathtt{E}_2) \leq \mu(m, z) + @_T(\mathtt{E}_1) + @_T(\mathtt{E}_2)$, we get the required result.

- $\mathtt{E} = \mathtt{E}_1, \mathtt{E}_2$. $T \vdash \mathtt{E} : (z, z')$ implies, using Rule (,), there exist z_1, z_2, z'_1, z'_2 such that $z = z_1 \| z_2, z' = z'_1 \| z'_2, T \vdash \mathtt{E}_1 : (z_1, z'_1)$ and $T \vdash \mathtt{E}_2 : (z_2, z'_2)$.

  Since $m = m_1 \| m_2$, with $m_1 = m[1, |z_1|]$ and similarly for m_2, we can assume, without loss of generality, that the minimum of the $\lfloor \underline{m[i]} / \sharp(z[i]) \rfloor$ in m occurs in m_1: thus $\mu(m_1, z_1) = \mu(m, z)$.

  By induction on $\mathtt{E}_1$, one gets $m'_1 = p_1(m_1) : z'_1$ and $|z'_1| = 0 \vee \mu(m'_1, z'_1) \leq \mu(m, z) + @_T(\mathtt{E}_1)$.

  Let c_2 be such that $\mu(m_2, z_2) = c_2$, with $c_2 \geq \mu(m, z)$ and $m_2 = m[|z_1| + 1, |z_1| + |z_2|]$. By induction on $\mathtt{E}_2$, one gets $m'_2 = p_2(m_2) : z'_2$ and $|z'_2| = 0 \ \vee \ \mu(m'_2, z'_2) \leq c_2 + @_T(\mathtt{E}_2)$.

  Since $m' = m'_1 \| m'_2$, then $|z'| = |z'_1| + |z'_2|$. So, either $|z'| = 0$ or

  $$\mu(m', z') = \min(\mu(m'_1, z'_1), \mu(m'_2, z'_2)) \ .$$

  One gets $\mu(m', z') \leq \mu(m, z) + @_T(\mathtt{E}_1) \leq \mu(m, z) + @_T(\mathtt{E})$, as required.

- $\mathtt{E} = \mathtt{E}_1 <: \mathtt{E}_2$. The proof is similar to the one for ":". Indeed, "<:" dispatches its input signals to its output signals, and then composes them, using ":". Since the dispatch operation does not modify the stream supports, this operation is, for rate correctness purposes, identical to ":".

- $\mathtt{E} = \mathtt{E}_1 :> \mathtt{E}_2$. Same as above, except that the dispatched signals are merged using the $+$ function. Since we know that $+$ is synchronous, mixing does not modify the rate behavior.

- $[\mathtt{L} >$ and $< \mathtt{L}]$. Record building and accessing are synchronous operations, as can be seen by looking at the rate of z and z'.

- $\mathtt{E} = \mathtt{E}_1 \sim \mathtt{E}_2$. $T \vdash \mathtt{E} : (z, \widehat{z'})$ implies, using Rule ($\sim$), there exist z_1, z_2, s'_2 such that $T \vdash \mathtt{E}_1 : (z_1, z')$, $T \vdash \mathtt{E}_2 : (z_2, z'_2)$, $z_2 = z'[1, |z_2|]$, $z'_2 = z_1[1, |z'_2|]$ and $z = z_1[|z'_2| + 1, |z_1|]$. One sees that $z_1 = z'_2 \| z$.

  Let $m' = fix(F)$, with $F = \lambda m'.p_1(p_2(@(m'[1, d_2]))\|m)$. We prove below that $P(m') = (|z'| = 0 \ \vee \ \mu(m', z') \leq c + @_T(\mathtt{E}))$, with $c = \mu(m, z)$, is true.

  Using fix point induction (which is valid since we stay in the cpo $\mathrm{Signal}^{|z'|}$), $P(m')$ needs to be proven for the bottom element $\perp$ and, assuming that P is true for m', show it is true for $F(m')$.

  - If $|z'| = 0$, in both steps, P is obviously true.
  - For the basis case of $\perp$, P is obvious too, since $\mu(\perp, z') = 0 \leq c + @_T(\mathtt{E})$, for all c.
  - Assume $\mu(m', z') \leq c + @_T(\mathtt{E})$. We need to show that $\mu(F(m'), z') \leq c + @_T(\mathtt{E})$, with $F(m') = p_1(p_2(@(m'[1, d_2]))\|m)$.

    Since $m'[1, d_2]$ is part of m', then $\mu(m'[1, d_2], z_2) = c_2$ with $c_2 \geq c$. By definition of the delaying semantics of `@`, which extends signal supports, then one has $\mu(@(m'[1, d_2]), z_2) = c_{@2}$ for some $c_{@2} \geq c_2$.

    By induction on $\mathtt{E}_2$, we get that $F(m') = p_1(m''\|m)$ with $\mu(m'', z'_2) \leq c_{@2} + @_T(\mathtt{E}_2)$ and m'' = $p_2(@(m'[1, d_2]))$.

    By concatenation of beams and impedances, one gets $\mu(m''\|m, z_1) = \min(\mu(m'', z'_2), \mu(m, z))$.

    By induction on $\mathtt{E}_1$, one has $\mu(F(m'), z') \leq \min(\mu(m'', z'_2), \mu(m, z)) + @_T(\mathtt{E}_1) \leq \mu(m, z) + @_T(\mathtt{E}_1)$, as required, since $@_T(\mathtt{E}_1) \leq @_T(\mathtt{E})$.
  - By fix point induction then, $\mu(fix(F), z') \leq c + @_T(\mathtt{E})$.

Since $\sharp(\widehat{t^\sharp}) = \sharp(t^\sharp)$ for any rated type $t^\sharp$, then $\mu(m', \widehat{z'}) = \mu(m, z) \leq c + @_T(\mathtt{E})$, as required. □

9. Conclusion

We provide the typing semantics, denotational semantics and correctness theorems for a new multirate extension of Faust, a functional programming language dedicated to musical, audio and more generally multimedia applications. We propose to link the introduction of record and vector datatypes in a synchronous setting to the presence of multiple signal rates. We describe a dedicated framework based on a new polymorphic dependent-type static semantics in which both vector sizes and rates are values, and prove a synchrony consistency theorem relating values and rates. This proposal is under implementation in the Faust compiler.

10. Acknowledgements

We thank Karim Barkati for his careful proofreading. This work is partially funded by the French ANR, as part of the ASTREE Project (2008 CORD 003 01).

References

[1] M. Puckette, The patcher, in: Proceedings of the International Computer Music Conference, ICMA, 1988.

[2] G. Assayag, C. Agon, OpenMusic Architecture, in: ICMA (Ed.), Proceedings of International Computer Music Conference, 1996, pp. 339–340.

[3] S. Letz, Y. Orlarey, D. Fober, The Role of Lambda-Abstraction in Elody, in: ICMA (Ed.), Proceedings of the International Computer Music Conference, 1998, pp. 377–384.

[4] E. Gaudrain, Y. Orlarey, A Faust Tutorial, Tech. rep., GRAME, Lyon (2003).

[5] H. Barendregt, The Lambda Calculus: Its Syntax and Semantics, North-Holland Publishing, 1981.

[6] Y. Orlarey, D. Fober, S. Letz, Syntactical and Semantical Aspects of Faust, Soft Computing 8 (9) (2004) 623–632.

[7] Y. Orlarey, D. Fober, S. Letz, An algebra for block diagram languages, in: ICMA (Ed.), Proceedings of International Computer Music Conference, 2002, pp. 542–547.

[8] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. L. Guernic, R. de Simone, The Synchronous Languages Twelve Years Later, in: Proceedings of the IEEE, Vol. 91, 2003.

[9] P. Caspi, M. Pouzet, A Functional Extension to Lustre, in: M. A. Orgun, E. A. Ashcroft (Eds.), International Symposium on Languages for Intentional Programming, World Scientific, Sydney, Australia, 1995.

[10] J.-L. Colaço, M. Pouzet, Clocks as first class abstract types, Embedded Software (2003) 134–155.

[11] L. Mandel, F. Plateau, Abstraction d'horloges dans les systèmes synchrones flot de données, in: Journées Francophones des Langages Applicatifs, 2009.

[12] M. Nebut, S. Pinchinat, A decidable clock language for synchronous specifications, in: Synchronous Languages, Applications, and Programming, Electronic Notes in Theoretical Computer Science, Grenoble, France, 2002.

[13] Y. Orlarey, Notes sur les extensions de Faust, Tech. rep., GRAME, Lyon (2009).

[14] J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science (vol. B): Formal Models and Semantics, MIT Press, 1990.

[15] G. Tzanetakis, G. Essl, P. Cook, Audio Analysis using the Discrete Wavelet Transform, in: Proc. Conf. in Acoustics and Music Theory Appl.. WSES, 2001.

$$\text{(i)}\ \frac{T(\mathtt{I}) \;=\; \Lambda l.(z,z') \qquad \forall (x,S)\in l \;.\; l'(x)\in S}{T \;\vdash\; \mathtt{I} : (z,z')[l'/l]}$$

$$(\subset)\ \frac{T \;\vdash\; \mathtt{E} : (z,z') \qquad z' \subset z'_1 \qquad z_1 \subset z}{T \;\vdash\; \mathtt{E} : (z_1,z'_1)}$$

$$(:)\ \frac{T \;\vdash\; \mathtt{E}_1 : (z_1,z'_1) \qquad T \;\vdash\; \mathtt{E}_2 : (z'_1,z'_2)}{T \;\vdash\; \mathtt{E}_1 : \mathtt{E}_2 : (z_1,z'_2)}$$

$$(,)\ \frac{T \;\vdash\; \mathtt{E}_1 : (z_1,z'_1) \qquad T \;\vdash\; \mathtt{E}_2 : (z_2,z'_2)}{T \;\vdash\; \mathtt{E}_1, \mathtt{E}_2 : (z_1 \| z_2, z'_1 \| z'_2)}$$

$$(<:)\ \frac{T \;\vdash\; \mathtt{E}_1 : (z_1,z'_1) \qquad T \;\vdash\; \mathtt{E}_2 : (z_2,z'_2) \qquad z'_1 \prec z_2}{T \;\vdash\; \mathtt{E}_1 <: \mathtt{E}_2 : (z_1,z'_2)}$$

$$(:>)\ \frac{T \;\vdash\; \mathtt{E}_1 : (z_1,z'_1) \qquad T \;\vdash\; \mathtt{E}_2 : (z_2,z'_2) \qquad z'_1 \succ z_2}{T \;\vdash\; \mathtt{E}_1 :> \mathtt{E}_2 : (z_1,z'_2)}$$

$$([>)\ \frac{|z| = |\mathtt{L}| \qquad iso(z) \qquad u = (\mathtt{L}, \|_z type)}{T \;\vdash\; [\mathtt{L} > : (z, (u^{\sharp(z)}))}$$

$$(<])\ \frac{u \;=\; (\mathtt{L},\mathtt{T}) \qquad \mathtt{L}' \subset \mathtt{L} \qquad z' \;=\; \|_{\mathtt{L}'}\ \lambda \mathtt{I}.\mathtt{T}[\mathtt{L}^{-1}(\mathtt{I})]^f}{T \;\vdash\; <\mathtt{L}'] : ((u^f), z')}$$

$$(\sim)\ \frac{T \;\vdash\; \mathtt{E}_1 : (z_1,z') \qquad T \;\vdash\; \mathtt{E}_2 : (z_2,z'_2) \qquad z_2 \;=\; z'[1,|z_2|] \qquad z'_2 \;=\; z_1[1,|z'_2|] \qquad iso(z_2)}{T \;\vdash \mathtt{E}_1 \sim \mathtt{E}_2 : (z_1[|z'_2|+1, |z_1|], \widehat{z'})}$$

Table 2: Faust Static Semantics

$$
\begin{array}{rcl}
E[\![\mathtt{I}]\!]r & = & r(\mathtt{I}) \\
E[\![\mathtt{E}_1 : \mathtt{E}_2]\!]r & = & p_2 \circ p_1 \\
E[\![\mathtt{E}_1 , \mathtt{E}_2]\!]r & = & \lambda m.p_1(m[1,d_1]) \| p_2(m[d_1+1, d_1+d_2]) \\
E[\![\mathtt{E}_1 <: \mathtt{E}_2]\!]r & = & \lambda m.p_2(\|_{1,d_2,d'_1} \lambda i.p_1(m)) \\
E[\![\mathtt{E}_1 :> \mathtt{E}_2]\!]r & = & \lambda m.p_2(\|_{1,d_2} \lambda i.\mathrm{mix}(p_1(m)[i, d'_1, d_2])) \\
 & & \text{where } \mathrm{mix}((s)) = (s) \text{ and } \mathrm{mix}((s)\|m) = E[\![+]\!]r((s)\|\mathrm{mix}(m)) \\
E[\![[\mathtt{L}>]\!]r & = & \lambda m.(\lambda t \in \bigcap_{i \in [1,|m|]} dom(m[i]).\lambda \mathtt{I} \in \mathtt{L}.m[\mathtt{L}^{-1}(\mathtt{I})](t)) \\
E[\![<\mathtt{L}']\!]r & = & \lambda (s).\|_{\mathtt{L}'} \lambda \mathtt{I}'.\lambda t.s(t)(\mathtt{I}') \\
E[\![\mathtt{E}_1 \sim \mathtt{E}_2]\!]r & = & \lambda m.fix(\lambda m'.p_1(p_2(@(m'[1,d_2]))\|m)) \\
 & & \text{where } @(m) = \|_m \lambda s.\mathrm{delay}(s, \lambda t.1)
\end{array}
$$

Table 3: Faust Denotational Semantics: we note $p_i = E[\![\mathtt{E}_i]\!]r$ and $(d_i, d'_i) = dim(p_i)$. We note "$\lambda x \in A.f(x)$" the function "$\lambda x.f(x)$ if $x \in A, \perp$ otherwise"