SEMANTICS FOR MULTIRATE FAUST

P. JOUVELOT[1], Y. ORLAREY[2]

Technical Report A/414, CRI, MINES ParisTech
ASTREE ANR Project D4.2 "Extensions pour Faust" Deliverable

Abstract. Faust is a functional programming language dedicated to the specification of executable monorate musical applications. We present here a multirate extension of the core of the Faust language, called MR Faust, together with a typing semantics, a denotational semantics and correctness theorems that link them together.

1. Introduction

From Music III, the first language for digital audio synthesis, developed by Max Mathews in 1959 at the Bell Labs, to Max [8], 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 [5], research in music programming languages has been very active since the early 60s. The computer music community has pioneered the field of end-user programming as well as the idea of using a programming language as a creative tool to invent complex objects like sounds, musical structures or real-time interactions. Nowadays the practice of Live Coding, with languages like ChucK [10], pushes even further the boundaries of what computer programming is by positioning this activity as a performing art. Moreover, with the convergence of digital arts, visual programming languages like Max have gained a large audience well outside the computer music community. Within this context,the programming language Faust [3] intends to provide a highly abstract, purely functional approach to signal processing while offering the highest level of performance. Faust aims at being complementary to existing audio languages by providing a viable and efficient alternative to C/C++ to develop 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 combine signal processors and (2) a macro language is used on top of this kernel to build and manipulate abstractions. 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 [4]). 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 [7] is monorate, and its semantics is defined in an operational framework. We propose here a multirate extension of Faust, along the lines of [6], together with its denotational definition, in order to offer a more abstract presentation of the language as well as the ability to eventually perform easier

Date: October 15, 2009.

mathematical proofs on Faust programs. We introduce and prove two main theorems: the Subject Reduction theorem ensures that the evaluation process preserves the typing properties of the language, while the Frequency Correctness theorem validates the multirate nature of this extension.

After this introduction, Section 2 provides a brief informal survey of Faust basic operations. Section 3 introduces the core Faust syntax. Section 4 is a proposal for a multirate extension, called MR Faust, of this core. Section 5 defines the static domains used to provide MR Faust static typing semantics (Section 6). Section 7 defines the semantic domains used in the MR Faust dynamic denotational semantics (Section 8). One needs to ensure that both static and dynamic semantics remain consistent along evaluation; this is the goal of the Subject Reduction theorem introduced and proved in Section 9. Showing that this multirate extension of Faust indeed behaves properly, i.e, that signals of different frequencies merge gracefully in a multirate program, is the subject of the Frequency Correctness theorem of Section 10. The last section concludes.

2. Overview of Faust

A Faust program does not describe a sound or a group of sounds, but a signal processor, something that gets input signals and produces output signals. The program source is organized as a set of definitions with at least the definition of the keyword `process` (the equivalent of `main` in C); running a Faust program amounts to plugging the I/O signals of `process` to the actual sound environment, such as a microphone and an audio system, for instance.

Here is a first Faust example that produces silence, i.e., a signal providing an infinite supply of 0s:

```
process = 0;
```

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

The second example is a little bit more sophisticated and copies the input signal to the output signal. It involves the `_` (underscore) primitive that denotes the identity function on signals (that is a simple audio cable for a sound engineer):

```
process = _;
```

Another very simple example is the conversion of a two-channel stereo signal into a one-channel mono signal using the `+` primitive that adds two signals together to yield a single, summed signal:

```
process = +;
```

Most Faust primitives are similar to their C counterparts on numbers, but lifted to signals. For example the Faust primitive `sin` operates on a signal s by applying the C function sin to each sample $s(t)$ of s; in other words, `sin` transforms an input signal s into an output signal s' such that $s'(t) = \sin(s(t))$. Yet, some signal processing primitives are specific to Faust. For example the delay operator `@` takes two input signals, s (the signal to be delayed) and d (the delay to be applied), and produces an output signal s' such that $s'(t) = s(t - d(t))$.

Contrarily to visual programming languages often used in the computer music world, where the user performs manual connections between operators to create so called block

diagrams, Faust primitives are assembled into block diagrams by using a set of high-level composition operations. You can think of these composition operators as a generalization of the mathematical function composition operator $\circ$.

Assume we want to connect the output of `+` to the input of `abs` in order to compute the absolute value of the output signal; this connection can be specified using the sequential composition operator '`:`' (colon):

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

Here is now an example of parallel composition (a stereo cable) using the operator '`,`' that puts in parallel its left and right expressions:

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

These operators can be arbitrarily combined. For example to multiply the input signal by 0.5, one can write:

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

Taking advantage of some syntactic sugar the details of which we will not address 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 block diagrams with delayed cycles. Here is the example of an integrator:

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

The `~` operator connects here in a feedback loop the output of `+` to the input of `_` (with an implicit 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 $s'(t) = s(t) + s'(t-1)$, thus performing a numerical integration operation

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

```
random  = +(12345) ~ *(1103515245);
noise   = random/2147483647.0;
process = noise*vslider("noise[style:knob]",0,0,100,0.1)/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, while `*(1103515245)` similarly denotes the multiplication of a signal by `1103515245`. These two operations are recursively composed using the `~` operator; this operator connects in a feedback loop the output of `+(12345)` to the input of `*(1103515245)` (with an implicit 1-sample delay) and the output of `*(1103515245)` to the input of `+(12345)`.

The definition of `noise` transforms the random signal into a noise signal by scaling it between -1.0 and +1.0. Finally, the definition of `process` adds a simple user interface to control the production of sound; 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 pseudonumbers (see the corresponding block diagram in Figure 1, where the little square denotes a 1-sample delay operator).

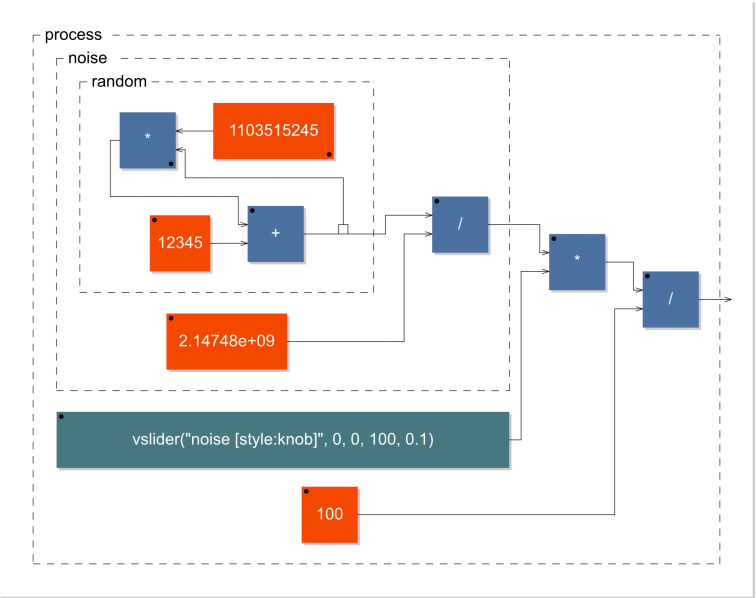

FIGURE 1. Graphic block diagram of the noise generator process.

3. Language Syntax

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

$$\begin{aligned} \mathtt{E} \quad ::= \quad & \mathtt{I} \mid \\ & \mathtt{E}_1 : \mathtt{E}_2 \mid \mathtt{E}_1, \mathtt{E}_2 \mid \\ & \mathtt{E}_1 <: \mathtt{E}_2 \mid \mathtt{E}_1 :> \mathtt{E}_2 \mid \\ & \mathtt{E}_1 \sim \mathtt{E}_2 \end{aligned}$$

We saw some examples of all these composition operators in Section 2, except for <: and :>; these operators perform fan-out and fan-in transformations, as we will see below.

In Faust, every expression represents a signal processor, i.e., a function that maps signals, which are functions from time to values, to other signals. Note that arithmetic operators do not appear as such in this syntax; they are considered as predefined identifiers and thus, for instance, "$\mathtt{I}_1 + \mathtt{I}_2$" is represented in this core version of Faust as "$\mathtt{I}_1, \mathtt{I}_2 : +$".

4. Multirate Extension

Faust, as described in [3], is a monorate language; in monorate languages, there is usually just one Time domain involved when accessing successive signal values. However, digital signal processing traditionally uses subsampling and oversampling operations heavily, which naturally lead to the introduction of multirate concepts. Since Faust targets a subset of DSP processing, the proposal introduced by Yann Orlarey [6] suggests

to use multiple frequencies to deal with such issues, instead of more general clocks, such as those present in traditional synchronous programming languages [1].

We propose to see clocking issues as an add-on to Faust. Frequencies f are elements of the $\text{Freq} = \mathbb{Q}^+$ domain. Signals, which are traditionnally typed according to the type of their codomain, are characterized by a pair, called a *rated type*, formed by a type and a frequency: $\text{Type}^\sharp = \text{Type} \times \text{Freq}$. Following [6], we posit that multiple rates in an application are introduced via vectors[1]. Vectors are created using the `vectorize` predefined primitive; informally, it collects n samples (the constant value n is provided by the signal that is the second argument to this primitive) from an input stream of frequency f and output vectors with n elements at frequency f/n. The `serialize` operation performs the reverse operation. Finally the processor primitive `[]`, which takes an input that includes a signal of vectors and one of integer indexes, is used to access elements of a vector.

One key and novel issue of this multirate extension is that the size of vectors are encoded into the vector type; moreover this size is provided via the *value* of a stream argument of the `vectorize` primitive. This calls for a dependent-type [9] static semantics that embeds values within types. We deal with this issue in the rest of this paper.

5. Static Domains

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

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

Since we mentioned that our type system will be a dependent-type semantics, 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; these pairs represent intervals that bound runtime values:

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

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 $a = (n, m)$ will be written $[n, m]$ in the sequel.

All base-typed expressions will be 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, will be typed using their size (the number n) and the type of their elements. Finally, since signed integers are part of types, via spans, we will need to perform some operations over these values, and thus introduce

[1] It is also suggested in [6] to introduce structures, an issue we leave here to further investigation.

the notion of type addition. The type domain is then:

$$\begin{aligned} t \in \text{Type} \quad = \quad & \text{Base} \times \text{Span} \mid \\ & \mathbb{N} \times \text{Type} \mid \\ & \text{Type} \times \text{Type} \end{aligned}$$

As a short hand, we note $b[a]$ for base types, $\text{vector}_n(t)$ for vector types and $t + t'$ for the addition of two types.

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 *wff*(t))**.** *A type t is* well-formed*, noted wff(t), iff:*

- *when $t = b[n, m]$, then $n \leq m$ and $\neg(n = m = -\omega)$ and $\neg(n = m = +\omega)$;*
- *when $t = \text{vector}_n(t')$, then wff(t');*
- *when $t = t' + t''$, then wff(t') and wff(t'').*

5.2. **Signals.** 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 frequencies f in Freq to manage rates:

$$f \in \text{Freq} \quad = \quad \mathbb{Q}^+$$

In our framework, the only signal processing operations we will perform that impact frequencies are related to over- and sub-sampling conversions. To represent such conversions, we will logically use multiplication and division arithmetic operations, which leads to the definition of Freq as the set of positive rational numbers.

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

$$\begin{aligned} t^\sharp \in \text{Type}^\sharp \quad = \quad & \text{Type} \times \text{Freq} \mid \\ & \text{Type}^\sharp \times \text{Type}^\sharp \end{aligned}$$

We will note t^f the rated type (t, f) and $t^\sharp + t'^\sharp$ the addition of two rated types. We also use simply t when f is not needed and there is no risk of confusion.

5.3. **Processors.** A MR Faust signal processor `E` maps ensembles (we called these *sheafs*) of signals to sheafs of signals. These sheafs 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:

$$z \in \text{Z} \quad = \quad \bigcup_n \text{Type}^{\sharp n}$$

The null impedance, in $\text{Type}^{\sharp 0}$, is (), and is used when no signal is present. A simple impedance is (t^f), and is the type of a sheaf containing one signal that maps time to values of type t at frequency f. The impedance length $|z|$ is defined such that $z \in \text{Type}^{\sharp |z|}$. 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 $\text{Type}^{\sharp d_1 + d_2}$ where $d_i = |z_i|$, defined as follows:

$$\begin{aligned} z[i] &= z_1[i] \ (1 \leq i \leq d_1) \\ z[i + d_1] &= z_2[i] \ (1 \leq i \leq d_2) \end{aligned}$$

To build more complex impedances, we introduce the $\|$ iterator as follows:

$$\begin{aligned} \|_{n,n',d} M &= (), \text{ if } n > n' \\ \|_{n,n',d} M &= M(n) \parallel \|_{n+d,n',d} M \text{ otherwise} \end{aligned}$$

where M is a function that maps integers to impedances. Intuitively, $\|_{n,n',d} M$ is the concatenation of $M(n), M(n+d), M(n+2d), ...M(n')$. As a short hand, $z[n, n', d]$, 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 $z[n, n'] = z[n, n', 1]$.

Definition 2 (Well-Formed Impedance *wff*(z))**.** *An impedance z is* well-formed*, noted* *wff*(z)*, iff, for all $i \in [1, |z|]$, there exist f, noted $\sharp(z[i])$, and t_i such that $z[i] = {t_i}^f$, with* *wff*(t_i)*.*

5.4. **Schemes.** A Core Faust signal processor maps input signals (a sheaf) to output signals (a sheaf). Since some processors are polymorphic (e.g., the identify processor _), the type of a processor must be a type scheme that contains both the input and output impedances, possibly abstracted over abstractable sorts S in Sort. Type schemes k in Scheme are defined as follows:

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

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

$$\begin{aligned} (z, z')[l'/l] &= (z[l'/l], z'[l'/l]) \\ z_1 || z_2 \ [l'/l] &= z_1[l'/l] || z_2[l'/l] \\ z[l'/()] &= z \\ z[l'/(x, S).l] &= z[l'(x)/x][l'/l] \\ x[v/x] &= v \\ y[v/x] &= y \end{aligned}$$

plus structural induction on types and frequencies.

Finally, typed identifiers are gathered in type environments T that map Ide to Scheme.

6. Static Semantics

The static semantics specifies how impedance pairs are assigned to signal processors. We first define some utilitary operations on static domains, and then provide static rules for MR Faust.

6.1. **Operations.** Complex MR 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: MR 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 $d_2 = |z_2|$:

$$\begin{aligned} z_1' \succ z_2 &= d_1' d_2 \neq 0 \text{ and} \\ &\quad \mathrm{mod}(d_1', d_2) = 0 \text{ and} \\ &\quad \sum_{i \in [0, d_1'/d_2 - 1]} z_1[1 + i d_2, (i+1) d_2] = z_2 \\ z_1' \prec z_2 &= d_1' d_2 \neq 0 \text{ and} \\ &\quad \mathrm{mod}(d_2, d_1') = 0 \text{ and} \\ &\quad \|_{1, d_2, d_1'} \lambda i. z_1' = z_2 \end{aligned}$$

where equality on impedances is defined by structural induction.

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 use the following static semantics addition rules:

(b+)
$$b[n, m] + b[n', m'] = b[n + n', m + m']$$

(v+)
$$\mathrm{vector}_n(t) + \mathrm{vector}_n(t') = \mathrm{vector}_n(t + t')$$

(t+)
$$\frac{t + t' = t''}{t^f + t'^f = t''^f}$$

(z+)
$$\frac{|z| = |z'| = |z''| \qquad \forall i \in [1, |z|].z[i] + z'[i] = z''[i]}{z + z' = z''}$$

The presence of values in types also induces an order relationship on Type, defined as follows:

Definition 3 (Subtype Ordering $t \subset t'$)**.** *A type t is a* subtype *of t', noted $t \subset t'$, if:*

- *either $t = t'$;*
- *or $t = b[n, m]$ and $t' = \mathrm{float}[n', m']$ and $[n, m] \subseteq [n', m']$;*
- *or $t = \mathrm{vector}_n(t_1)$ and $t' = \mathrm{vector}_n(t_1')$ and $t_1 \subset t_1'$.*

Note that this order relation is not defined on non-reductible sum types.

6.2. **Rules.** Faust is strongly and statically typed. Every expression, a signal processor, is typed by its input and output impedances:

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

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 of the same symbol.

We assume that there is an initial type environment T_0 that satisfies the following definitions for predefined signal processors:

$$\begin{aligned}
T_0(\texttt{_}) &= \Lambda t^\sharp : \text{Type}^\sharp.((t^\sharp), (t^\sharp)) \\
T_0(\texttt{!}) &= \Lambda t^\sharp : \text{Type}^\sharp.((t^\sharp), ()) \\
T_0(\texttt{0}) &= \Lambda b : \text{Base}.f : \text{Freq}.((), (b[0,0]^f)) \\
T_0(\texttt{-2.8}) &= \Lambda f : \text{Freq}.((), (\text{float}[-3,-2]^f)) \\
T_0(\texttt{+}) &= \Lambda t^\sharp : \text{Type}^\sharp.t'^\sharp : \text{Type}^\sharp.((t^\sharp, t'^\sharp), (t^\sharp + t'^\sharp)) \\
T_0(\texttt{@}) &= \Lambda f : \text{Freq}.t^\sharp : \text{Type}^\sharp.m : \mathbb{N}.((t^\sharp, \text{int}[1,m]^f), (t^\sharp)) \\
T_0(\texttt{vectorize}) &= \Lambda f : \text{Freq}.f' : \text{Freq}.t : \text{Type}.n : \mathbb{N}.((t^f, \text{int}[n,n]^{f'}), (\text{vector}_n(t)^{f/n})) \\
T_0(\texttt{[]}) &= \Lambda f : \text{Freq}.t : \text{Type}.n : \mathbb{N}.((\text{vector}_n(t)^f, \text{int}[0,n-1]^f), (t^f)) \\
T_0(\texttt{serialize}) &= \Lambda f : \text{Freq}.t : \text{Type}.n : \mathbb{N}.((\text{vector}_n(t)^f), (t^{f*n}))
\end{aligned}$$

Note that numerical operators (which are also signal procesors) must be able to deal with any type, and are as such associated to a polymorphic type scheme in the type environment; this is a consequence of the implicit mixing introduced by the lax impedance matching relation $\succ$, as we will see. A similar requirement exists for constants such as 0 (which are predefined identifiers, anyway).

The inference rules are defined below:

(ide)

$$\frac{\begin{array}{rcl} T(\texttt{I}) & = & \Lambda l.(z, z') \\ \forall (x, S) \in l & . & l'(x) \in S \end{array}}{T \vdash \texttt{I} : (z, z')[l'/l]}$$

(seq)

$$\frac{\begin{array}{c} T \vdash \texttt{E}_1 : (z_1, z_1') \\ T \vdash \texttt{E}_2 : (z_1', z_2') \end{array}}{T \vdash \texttt{E}_1 : \texttt{E}_2 : (z_1, z_2')}$$

(par)

$$\frac{\begin{array}{c} T \vdash \texttt{E}_1 : (z_1, z_1') \\ T \vdash \texttt{E}_2 : (z_2, z_2') \end{array}}{T \vdash \texttt{E}_1, \texttt{E}_2 : (z_1 \| z_2, z_1' \| z_2')}$$

(split)

$$\frac{\begin{array}{rcl} T & \vdash & \mathtt{E}_1 : (z_1, z_1') \\ T & \vdash & \mathtt{E}_2 : (z_2, z_2') \\ z_1' & \prec & z_2 \end{array}}{T \vdash \mathtt{E}_1 <: \mathtt{E}_2 : (z_1, z_2')}$$

(merge)

$$\frac{\begin{array}{rcl} T & \vdash & \mathtt{E}_1 : (z_1, z_1') \\ T & \vdash & \mathtt{E}_2 : (z_2, z_2') \\ z_1' & \succ & z_2 \end{array}}{T \vdash \mathtt{E}_1 :> \mathtt{E}_2 : (z_1, z_2')}$$

(loop)

$$\frac{\begin{array}{rcl} T & \vdash & \mathtt{E}_1 : (z_1, z') \\ T & \vdash & \mathtt{E}_2 : (z_2, z_2') \\ z_2 & = & z'[1, |z_2|] \\ z_2' & = & z_1[1, |z_2'|] \end{array}}{T \vdash \mathtt{E}_1 \sim \mathtt{E}_2 : (z_1[|z_2'|+1, |z_1|], \widehat{z'})}$$

(sub)

$$\frac{\begin{array}{rcl} T & \vdash & \mathtt{E} : (z, z') \\ z' & \subset & z_1' \\ z_1 & \subset & z \end{array}}{T \vdash \mathtt{E} : (z_1, z_1')}$$

Some rules are rather straightforward. Rule (ide) 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`. In Rule (seq), signal processors are plugged in sequence, which requires that the output impedance of $\mathtt{E}_1$ is the same as $\mathtt{E}_2$'s input. In Rule (par), running two signal processors in parallel requires that their input and output impedances are concatenated. In Rules (split) and (merge), the $\prec$ and $\succ$ constraints are used to ensure that a proper mixing of the output of $\mathtt{E}_1$ to the input of $\mathtt{E}_2$ is possible.

The most involved rule is (loop). Here, the input impedance z_2 of the feedback expression $\mathtt{E}_2$ 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 $z_1[|z_2'|+1, |z_1|]$. 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 z, noted $\widehat{z}$, is such that $|\widehat{z}| = |z|$ and:*

- $\forall i \in [1, |z|].\widehat{z}[i] = \widehat{z[i]}$;
- $\widehat{\mathrm{vector}_n(t)^f} = \mathrm{vector}_n(\widehat{t})^f$;
- $\widehat{b[a]^f} = b[\widehat{a}]^f$;

- $\widehat{[n,m]} = [-\omega, +\omega]$.

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

Finally, Rule (sub) allows types to be extended according to the order relationship induced by spans in types and basic types.

7. Dynamic Domains

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

$$t \in \text{Time} \quad = \quad \mathbb{N}$$

Signals map times to values v in Val :

$$\begin{aligned} v \in \text{Val} &= N + R + \bigcup_n \text{Val}^n \\ N &= \mathbb{N} + \{\bot\} + \{?\} \\ R &= \mathbb{R} + \{\bot\} + \{?\} \end{aligned}$$

Since the evaluation process may be non-terminating, we posit that Val is a cpo, with order relation $\sqsubset$ and bottom $\bot$; all operations in Val are strict. 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 $\bot$, we assume $o(?, v) =?$. For a vector $v \in \text{Val}^n$, we define its size $|v|$ by $v \in \text{Val}^{|v|}$.

A signal s, which is a history denoted by a function, is a member of Signal, with:

$$s \in \text{Signal} \quad = \quad \text{Time} \rightarrow \text{Val}$$

We define the domain $dom(s)$ of a signal s by $dom(s) = \{t / s(t) \neq \bot\}$. The size of this domain $|dom(s)|$, called its support $\underline{s}$, is a member of $\mathbb{N} + \{\omega\}$, where ω is used to deal with infinite signals. 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 sheafs $m = (m_1, ..., m_n)$ in Sheaf:

$$m \in \text{Sheaf} \quad = \quad \bigcup_n \text{Signal}^n$$

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

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

At last, a processor p in Proc is the basic bloc of a Faust program:

$$p \in \text{Proc} \quad = \quad \text{Sheaf} \rightarrow \text{Sheaf}$$

We define $dim(p) = (n, n')$ such that $p \in \text{Signal}^n \rightarrow \text{Signal}^{n'}$.

The standard semantics of a Faust expression is a function of the semantics of its free identifiers; we collect these in a a state r, a member of State, with:

$$r \in \text{State} \quad = \quad \text{Ide} \rightarrow \text{Proc}$$

8. Denotational Semantics

We assume given an initial state r_0 such that:

$$\begin{aligned}
r_0(\texttt{_}) &= \lambda(s).(s) \\
r_0(\texttt{!}) &= \lambda(s).() \\
r_0(\texttt{0}) &= \lambda().(\lambda t.0) \\
r_0(\texttt{+}) &= \lambda(s_1, s_2).(\lambda t.s_1(t) + s_2(t)) \\
r_0(\texttt{/}) &= \lambda(s_1, s_2). \\
&\quad (\lambda t. \\
&\quad\ s_1(t)/s_2(t) \text{ if } t < \min(\underline{s_1}, \underline{s_2}) \text{ and } s_2(t) \neq 0, \\
&\quad\ ? \text{ if } t < \min(\underline{s_1}, \underline{s_2}), \\
&\quad\ \bot \text{ otherwise}) \\
r_0(\texttt{vectorize}) &= \lambda(s_1, s_2).(\lambda t.(s_1(nt), ..., s_1(n-1+nt)) \text{ where } n = s_2(0)) \\
r_0(\texttt{[]}) &= \lambda(s_1, s_2).(\lambda t.s_1(t)[s_2(t)]) \\
r_0(\texttt{serialize}) &= \lambda(s). \\
&\quad (\lambda t. \\
&\quad\ ? \text{ if } s(0) = 0, \\
&\quad\ s(\lfloor t/|s(0)| \rfloor)[\text{mod}(t, |s(0)|)] \text{ otherwise}) \\
r_0(\texttt{@}) &= \lambda(s_1, s_2).(\text{delay}(s_1, s_2))
\end{aligned}$$

where we assumed the existence of the delay function defined as:

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

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 is thus $\text{delay}(s_1, \lambda t.1)$.

These definitions assume that $T \vdash \texttt{0} : t$ for all types t, since this is needed for the definition of delay to make sense. Similarly `+` is supposed to be defined for all types since it is used in the definition of :> (see below).

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`*.*

We now define the semantic function E:

$$E \in \text{Exp} \rightarrow \text{State} \rightarrow \text{Sheaf} \rightarrow \text{Sheaf}$$

The semantics $E[\![\texttt{E}]\!]r$ of an expression `E` in a state r such that $T \vdash r$ is a function that maps an input sheaf m to an output sheaf m'. In the following definitions, we note $p_i = E[\![\texttt{E}_i]\!]r$ and $(d_i, d'_i) = dim(p_i)$:

$$
\begin{aligned}
E[\![\mathtt{I}]\!]r &= r(\mathtt{I}) \\
E[\![\mathtt{E}_1 : \mathtt{E}_2]\!]r &= p_2 \circ p_1 \\
E[\![\mathtt{E}_1, \mathtt{E}_2]\!]r &= \lambda m.p_1(m[1,d_1]) \| p_2(m[d_1+1, d_1+d_2]) \\
E[\![\mathtt{E}_1 <: \mathtt{E}_2]\!]r &= \lambda m.p_2(\|_{1,d_2,d_1'} \lambda i.p_1(m)) \\
E[\![\mathtt{E}_1 :> \mathtt{E}_2]\!]r &= \lambda m.p_2(\|_{1,d_2,1} \lambda i.\mathrm{sum}(p_1(m)[i, d_1', d_2])) \\
&\quad \text{where } \mathrm{sum}((s)) = (s) \text{ and } \mathrm{sum}((s)\|m) = r(\mathtt{+})((s)\|\mathrm{sum}(m)) \\
E[\![\mathtt{E}_1 \sim \mathtt{E}_2]\!]r &= \lambda m.fix(\lambda m'.p_1(p_2(@(m'[1,d_2]))\|m)) \\
&\quad \text{where } @(()) = () \text{ and } @((s)\|m) = (\mathrm{delay}(s, \lambda t.1))\|@(m)
\end{aligned}
$$

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 sheaf of size at least $d_1 + d_2$ and feeds the first d_1 signals into p_1 and the subsequent d_2 into p_2; the outputs are concatenated. The split construct repeatedly concatenates the outputs of p_1 to feed into the (larger) d_2 inputs of p_2. The merge construct performs a kind of opposite operation; all $\mathrm{mod}(i, d_2)$-th output values of p_1 are summed together to construct the i-th input value of p_2. The loop expression is, as can be expected, the most complex one. 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 version of m'.

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

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

9. 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 sheafs are preserved.

Definition 7 (Value Type Correctness $v : t$)**.** *A value* v *is* type correct*, noted* $v : t$*, iff:*

- *when* $v \in \mathbb{N}$*, then* $t = \mathrm{int}[n, m]$ *and* $n \leq v \leq m$*;*
- *when* $v \in \mathbb{R}$*, then* $t = \mathrm{float}[n, m]$ *and* $n \leq v \leq m$*;*
- *when* $v \in \bigcup_n \mathrm{Val}^n$*, then* $t = \mathrm{vector}_n(t'), n = |v|$ *and, for all* $i \in [0, n-1], v[i] : t'$.

Definition 8 (Signal Type Correctness $s : t^f$)**.** *A signal* s *is* type correct *w.r.t. a type* t^f*, noted* $s : t^f$*, if, for all* $u \in dom(s)$*, one has* $s(u) : t$.

Definition 9 (Sheaf Type Correctness $m : z$)**.** *A sheaf* 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 10 (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 MR Faust expression $\mathtt{E}$, if the environment T and state r are consistent and $\mathtt{E}$ maps sheafs of impedance z to sheafs of impedance z', then, given a sheaf m that is type correct w.r.t. z, then the semantics $p(m)$ of $\mathtt{E}$ will yield a sheaf m' of impedance z':

Theorem 2 (Subject Reduction)**.** *For all $\mathtt{E}, T, z, z', r$ and m, if*

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

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 (seq), 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 (par), 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 Sheaf 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 $\mathtt{E}_1$, 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 $\mathtt{E}_2$, $E[\![\mathtt{E}_2]\!]r(m[d_1 + 1, d_1 + |z_2|]) : z'_2$ and $dim(p_2) = (|z_2|, |z'_2|)$. Since $d_2 = |z_2|$, then $E[\![\mathtt{E}_2]\!]r(m[d_1 + 1, d_1 + d_2]) : z'_2$.
  The definition of E on $\mathtt{E}$ yields $p(m) = p_1(m[1, d_1]) \| p_2(m[d_1 + 1, d_1 + d_2])$. By definition of Sheaf Type Correctness, $p(m) : z'_1 \| z'_2 = z'$ and $dim(p) = (|m|, |z'|) = (|z|, |z'|)$.
- $\mathtt{E} = \mathtt{E}_1 <: \mathtt{E}_2$. $T \vdash \mathtt{E} : (z, z')$ implies, using (split), there exist z'_1, z_2, k such that $T \vdash \mathtt{E}_1 : (z, z'_1)$, $T \vdash \mathtt{E}_2 : (z_2, z')$, $|z_2| = k|z'_1|$ and, for all i in $[0, k - 1]$, $z_2[1 + i|z'_1|, |z'_1| + i|z'_1|] = z'_1$.
  By induction on $\mathtt{E}_1$, one gets $p_1(m) : z'_1$ and $dim(p_1) = (|z|, |z'_1|)$. By induction on $\mathtt{E}_2$, $dim(p_2) = (|z_2|, |z'|)$. 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 Sheaf Type Correctness and k, one gets $m' : z_2$. By induction on $\mathtt{E}_2$, one gets $p_2(m') : z'$ and $dim(p_2) = (|z_2|, |z'|)$.

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

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

  By induction on $\mathtt{E}_1$, one gets $p_1(m) : z'_1$ and $dim(p_1) = (|z|, |z'_1|)$. By induction on $\mathtt{E}_2$, $dim(p_2) = (|z_2|, |z'|)$. 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:

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

  Thus, by definition of *sum* and the addition of impedances, one gets:

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

  Let $m_2 = \|_{1,d_2,1} \lambda i.sum(m^i)$. 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 $dim(p) = (|z|, |z'|)$.

- $\mathtt{E} = \mathtt{E}_1 \sim \mathtt{E}_2$. $T \vdash \mathtt{E} : (z, \widehat{z'})$ implies, using (loop), 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 $\text{Signal}^{|z'|}$), 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 $\text{Signal}^{|z'|}$: $\perp' = (\lambda t.\ \perp, ..., \lambda t.\ \perp) : z'$. 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 $\mathtt{E}_1$, 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 (sub) 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 3 (Initial State Type Consistency)**.**

$$\vdash T_0, r_0$$

10. Frequency Correctness Theorem

One needs, in the presence of signals of different frequencies, to ensure the consistency of the frequency assignment to signals. In particular, we show below that one can bound the support of signals (and, more generally, sheafs) in a way consistent with their relative frequencies; this is the Frequency Correctness theorem.

Definition 11 (Sheaf Boundness $(m, z)\ !\ c$)**.** *For any $c \in \mathbb{Q}^+$, a sheaf m of impedance z is* c-bounded*, noted $(m, z)\ !\ c$, if* $\min_{i\in[1,|z|]}(\underline{m[i]}/\sharp(z[i])) \leq c$.

Informally, when $(m, z)\ !\ c$, then there is at least one signal i^* in m that has $c\sharp(z[i^*])$ elements or less in its domain of definition[2]. This is interesting since the supports of signals in a sheaf 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 computation (all subsequent values are $\perp$), thus yielding some information about the relative size of buffers needed to perform it.

Another way to look at c-boundness comes from c itself; being the inverse of a frequency, its unit is the second, and thus c is a time. The definition of Sheaf Boundness yields an upper bound on the time required to exhaust (at least one of) the signals of m, thus providing a 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.

The Frequency Correctness theorem (see below) states that, given given a MR Faust expression `E` (`@` excluded, for reasons we address below), if the environment T and state r are consistent and `E` maps sheafs of impedance z to sheafs of impedance z', then, given a sheaf m that is type correct w.r.t. z and is c-bounded, then the semantics $p(m)$ of `E` will yield a c-bounded sheaf m' of impedance z'. Basically, this theorem tells us that an upper-bound of the running time of `E` is always the same, whichever way we try to assess it via any of its observable facets (namely input or output data): c is consistent and thus a characteristics of `E`.

Theorem 4 (Frequency Correctness)**.** *For all* `E` *not containing* `@`*, T, z, z', c, r, m and m', if*

$$\begin{aligned} &\vdash T, r \\ &m : z \\ &(m, z)\ !\ c \\ &T \vdash \texttt{E} : (z, z'), \end{aligned}$$

then $|z'| = 0\ \vee\ (m', z')\ !\ c$, where $m' = p(m) : z'$ and $p = E[\![\texttt{E}]\!]r$.

Proof. By induction on the structure of `E`.

- `E` = `I`. Use $E[\![\texttt{I}]\!]r = r(\texttt{I})$ and then:
  - trivial for `_`;

[2]When signals are properly synchronized, e.g., in an actual computation, all $\underline{m[i]}/\sharp(z[i])$ are equal, and the comments in these paragraphs apply to all signals.

- for constants (thus with $z = ()$), since the minimum of the empty set is ω, then $c = \omega$. The property $(m', z')\ !\ \omega$ is satisfied, since there is no limit on the number of constant values for the output signal;
- for `!`, since $|z'| = 0$, then the theorem is trivially satisfied;
- for synchronous operations such as `+` or `[]`, this is obvious since $\sharp(z[i]) = \sharp(z'[i'])$. Note that we would need to use $c' > c$ as an upper-bound instead of c to deal with the delay operator `@`;
- for vectorizing and serializing operations, the relationship on frequencies is, by design, inverse of the one on the size of the domains, thus yielding the expected relation.

• $\mathtt{E} = \mathtt{E}_1 : \mathtt{E}_2$. $T \vdash \mathtt{E} : (z, z')$ implies, using (seq), 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 $\mathtt{E}_1$, $m'_1 = p_1(m) : z'_1$ and $(m'_1, z'_1)\ !\ c$.
By induction on $\mathtt{E}_2$, one gets $m' = p_2(m'_1) : z'$ and $(m', z')\ !\ c$, as required.

• $\mathtt{E} = \mathtt{E}_1, \mathtt{E}_2$. $T \vdash \mathtt{E} : (z, z')$ implies, using (par), 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 $\underline{m[i]}/\sharp(z[i])$ in m occurs in m_1: thus $(m_1, z_1)\ !\ c$.
By induction on $\mathtt{E}_1$, one gets $m'_1 = p_1(m_1) : z'_1$ and $(m'_1, z'_1)\ !\ c$.
Let c_2 be such that $(m_2, z_2)\ !\ c_2$, with $c_2 \geq c$ 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 $(m'_2, z'_2)\ !\ c_2$.
Since $m' = m'_1 \| m'_2$, then $(m', z')\ !\ \min(c, c_2) = c$, 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 frequency correctness purposes, identical to ":".

• $\mathtt{E} = \mathtt{E}_1 :> \mathtt{E}_2$. Same as above, except that the dispatched signals are merged using addition; since addition is a synchronous operation, this does not modify the frequency behavior.

• $\mathtt{E} = \mathtt{E}_1 \sim \mathtt{E}_2$. $T \vdash \mathtt{E} : (z, \widehat{z'})$ implies, using (loop), 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 $(m', z')\ !\ c$. 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 $\mathrm{Signal}^{|z'|}$: $\perp' = (\lambda t.\ \perp, ..., \lambda t.\ \perp) : z'$. Since $\underline{\lambda t.\ \perp} = 0$, then all supports in $\perp'$ are 0, and one just needs to prove that $0 \leq c$, which is always true.
- Assume $(m', z')\ !\ c$. We need to show that $(F(m'), z')\ !\ c$. One has $F(m') = p_1(p_2(@(m'[1, d_2])) \| m)$.
By definition of the delaying semantics of `@`, if $(m', z')\ !\ c$, then one has $(@(m'), z')\ !\ c + \min_{i \in [1, |z'|]}(1/\sharp(z'[i]))$; this is also valid for $(@(m'[1, d_2]), z')$.

By induction on $\mathtt{E}_2$, $F(m') = p_1(m''\|m)$, where $(m'', z'_2 = z_1[1, |z'_2|]) \mathbin{!} c + \min_{i\in[1,|z'|]}(1/\sharp(z'[i]))$.
Since $(m, z = z_1[|z'_2| + 1, |z_1|]) \mathbin{!} c$, then, by concatenation of sheafs and impedances, $(m''\|m, z_1) \mathbin{!} \min(c, c + \min_{i\in[1,|z'|]}(1/\sharp(z'[i]))) = c$. By induction on $\mathtt{E}_1$, one has $(F(m'), z') \mathbin{!} c$, as required.

– By fix point induction then, $(m', z') \mathbin{!} c$. Since $\sharp(\widehat{t^\sharp}) = \sharp(t^\sharp)$ for any rated type, then $(m', \widehat{z'}) \mathbin{!} c$, as required. □

11. Conclusion

We provide the typing semantics, denotational semantics and static correctness theorems for MR Faust, a multirate extension, sketched in [6], of Faust, a functional programming language dedicated to musical applications. The need for multiple signal rates is linked to the introduction of vector data in a synchronous setting. We propose a dedicated framework based on a new polymorphic dependent-type type system in which both vector sizes and frequencies are values, and prove two consistent theorems for both values and frequencies.

12. Acknowledgements

This work is partially funded by the French Agence nationale de la recherche, as part of the ASTREE Project (2008 CORD 003 01).

References

[1] A.Benveniste, P.Caspi, S.A.Edwards, N.Halbwachs, P.Le Guernic, and R.de Simone. The synchronous languages twelve years later. In *Proceedings of the IEEE*, volume 91, January 2003.
[2] G. Assayag and C. Agon. OpenMusic Architecture. In ICMA, editor, *Proceedings of International Computer Music Conference*, pages 339–340, 1996.
[3] E. Gaudrain and Y. Orlarey. A Faust Tutorial. Technical report, GRAME, Lyon, 2003.
[4] H.P.Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland Publishing, 1981.
[5] S. Letz, Y. Orlarey, and D. Fober. The Role of Lambda-Abstraction in Elody. In ICMA, editor, *Proceedings of the International Computer Music Conference*, pages 377–384, 1998.
[6] Y. Orlarey. Notes sur les extensions de Faust. Technical report, GRAME, Lyon, 2009.
[7] Y. Orlarey, D. Fober, and S. Letz. Syntactical and semantical aspects of faust. *Soft Computing*, 8(9):623–632, 2004.
[8] M. Puckette. The patcher. In *Proceedings of the International Computer Music Conference*. ICMA, 1988.
[9] Jan van Leeuwen, editor. *Handbook of theoretical computer science (vol. B): formal models and semantics*. MIT Press, Cambridge, MA, USA, 1990.
[10] Ge Wang and Perry R. Cook. Chuck: A concurrent, on-the-fly, audio programming language. In ICMA, editor, *Proceedings of International Computer Music Conference*, 2003.

[1]CRI, Maths et systemes, MINES ParisTech; [2]GRAME, Lyon
E-mail address: pierre.jouvelot@mines-paristech.fr, orlarey@grame.fr