DEPENDENT VECTOR TYPES FOR MULTIRATE FAUST

Pierre Jouvelot
CRI, Mathématiques et systèmes, MINES ParisTech
pierre.jouvelot@mines-paristech.fr

Yann Orlarey
Grame
orlarey@grame.fr

ABSTRACT

Faust is a functional programming language dedicated to the specification of executable monorate synchronous musical applications. To extend Faust capabilities to domains such as spectral processing, we introduce here a multirate extension of the core Faust language. The key 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. This interplay between vectors 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 a correctness theorem that show that this extension preserves the language synchonous characteristics. This new design is under implementation in the Faust compiler.

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 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. Since Faust current definition does not offer first-class vectors, this proposal kills two birds with one stone by adding both multirate processing and vector data structures; this interplay between vectors 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 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) a Frequency Correctness theorem that validates the multirate synchronous nature of this vector extension.

After this introduction, Section 2 provides a brief informal survey 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; 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. 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 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 and an audio system, for instance.

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 simple example is the conversion of a two-channel stereo signal into a one-channel mono 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, generalizations of the mathematical function composition operator ∘. 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 (a stereo cable) using the operator '`,`' that puts in parallel its left and right expressions. It 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. 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 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 $s'(t) = s(t) + s'(t-1)$, 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 : /;
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, 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 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 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 pseudo random numbers (see the familiar block diagram representation of this `process` in Figure 1, where the little square near the addition block denotes a 1-sample delay operator).

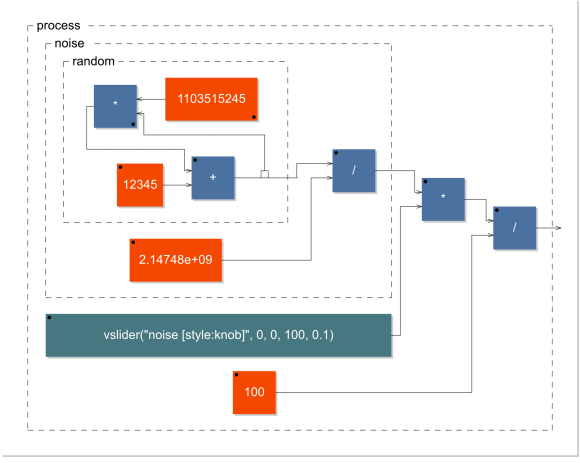

Figure 1. Noise generator `process` block diagram

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

Faust, as described in [4], 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 a subset of DSP processing, the proposal introduced by Yann Orlarey [7] suggests to use multiple frequencies to deal with such issues, instead of more general clocks, such as those present in traditional synchronous programming languages [8]. We informally describe below this approach, and illustrates it with a simple example of its use.

3.1 Frequencies for vector processing

We propose to see clocking issues as an add-on to the Faust static semantics (Faust is a strongly typed language). Frequencies f are elements of the $\text{Freq} = \mathbb{Q}^+$ 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 frequency: $\text{Type}^\sharp = \text{Type} \times \text{Freq}$.

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 frequency f and outputs vectors with n elements at frequency f/n; if the input values are of type t, then output vector samples have type $\text{vector}_n(t)$. The dual `serialize` primitive maps a signal of vectors of type $\text{vector}_n(t)$ at frequency f to the signal of frequency $f * n$ of their linearized elements, of type t. 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 # builds a signal of concatenated vectors from its two vector signal inputs.

The second key feature of this multirate extension is thus that the size n of vectors are 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 [9] 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. Before describing formally our framework in the remainder of this paper, we illustrate it with an example.

3.2 Haar Filtering, an Example

To get a better intuitive understanding of how these vector constructs interact with Faust primitives, we present a Haar-like downsampling process, a simplified step in the Discrete Wavelet Transform shown to be of use, for instance, in some audio feature extraction algorithms [10]. The signal processor `process` takes an input signal s at frequency f and produces two output signals, the mean o_1 and difference o_2, at frequency $f/2$, 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);
left    = _,!;
process =
       _ <: (mean:down),down <: left,-;
```

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 frequency f into an output signal of frequency $f/2$, 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 averaged, 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 signals are then fanned-out into the four-input signal processor `left,-`; it indeed takes four inputs, since (1) `left` takes a pair of signals, here (m_2, s_2), keeping only its left component m_2 using the primitive `!` that maps, by definition, its own s_2 to nothing and (2) the substraction operation – takes two inputs, here again m_2 and s_2, yielding the signal $m_2 - s_2$. 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:

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

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:

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

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 $a = (n, m)$ is 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 the notion of type addition. The type domain is then [1] :

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

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 $\mathit{wff}(t)$**)**
A type t is well-formed*, noted $\mathit{wff}(t)$, 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 $\mathit{wff}(t')$ and $n \geq 0$;*
- *when $t = t' + t''$, then $\mathit{wff}(t')$ and $\mathit{wff}(t'')$.*

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 frequencies f in Freq to manage rates:

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

In our framework, the only signal processing operations that impact frequencies are related to over- and sub-sampling conversions. To represent such conversions, we use multiplication and division arithmetic operations, thus defining Freq 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 frequencies. We link these two concepts in the notion [2] 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.

[1] The use of the same symbol, t, for both times and types should not be confusing, since they operate in different semantics.

[2] The notation $\sharp$ is, of course, different from the vector concatenation Faust primitive.

4.3 Impedances

A Faust signal processor maps sets (we called these *beams*) 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:

$$z \in \text{Z} \quad = \quad \bigcup_{n \geq 0} \text{Type}^{\sharp n}$$

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 beam 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:

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

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

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

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

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:

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

For readability [3], we note $\Lambda x : S ... x' : S'.(z, z')$ the scheme $(((x, S), ..., (x', S')), z, z')$, where x are abstracting variables in Var. These schemes will be instantiated

[3] 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.

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 thus defined as follows:

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

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.

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 $d_2 = |z_2|$:

$$\begin{array}{rcl} z_1' \succ z_2 & = & d_1' d_2 \neq 0 \text{ and} \\ & & \mathrm{mod}(d_1', d_2) = 0 \text{ and} \\ & & \displaystyle\sum_{i \in [0, d_1'/d_2 - 1]} z_1[1 + id_2, (i+1)d_2] = z_2 \\ z_1' \prec z_2 & = & d_1' d_2 \neq 0 \text{ and} \\ & & \mathrm{mod}(d_2, d_1') = 0 \text{ and} \\ & & \|_{1, d_2, d_1'} \lambda i. z_1' = z_2 \end{array}$$

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:

$$\begin{array}{lrcl} \text{(b+)} & b[n, m] + b[n', m'] & = & b[n + n', m + m'] \\ \text{(v+)} & \mathrm{vector}_n(t) + \mathrm{vector}_n(t') & = & \mathrm{vector}_n(t + t') \end{array}$$

The presence of values in types also induces a natural order relationship $t \subset t'$ on Type.

5.3 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, $T_0(\_) = \Lambda t^\sharp : \mathrm{Type}^\sharp.((t^\sharp), (t^\sharp))$ and $T_0(+) = \Lambda t^\sharp : \mathrm{Type}^\sharp . t'^\sharp : \mathrm{Type}^\sharp.((t^\sharp, t'^\sharp), (t^\sharp + t'^\sharp))$. 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 frequency, a constraint enforced by the use of the same $t^\sharp$ in these type schemes. A similar requirement exists for constants such as `0` (which are too predefined identifiers in T_0).

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

- $T_0(\mathtt{vectorize}) =$
  $\Lambda f : \mathrm{Freq}. f' : \mathrm{Freq}. t : \mathrm{Type}. n : \mathbb{N}.$
  $((t^f, \mathrm{int}[n, n]^{f'}), (\mathrm{vector}_n(t)^{f/n}))$;
- $T_0(\#) =$
  $\Lambda f : \mathrm{Freq}. t : \mathrm{Type}. m : \mathbb{N}. n : \mathbb{N}.$
  $((\mathrm{vector}_m(t)^f, \mathrm{vector}_n(t)^f), (\mathrm{vector}_{m+n}(t)^f))$;
- $T_0(\mathtt{[]}) =$
  $\Lambda f : \mathrm{Freq}. t : \mathrm{Type}. n : \mathbb{N}.$
  $((\mathrm{vector}_n(t)^f, \mathrm{int}[0, n-1]^f), (t^f))$;
- $T_0(\mathtt{serialize}) =$
  $\Lambda f : \mathrm{Freq}. t : \mathrm{Type}. n : \mathbb{N}.((\mathrm{vector}_n(t)^f), (t^{f*n}))$.

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 frequency 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 frequencies; this is key to prove the theorem of Section 6.3.

5.4 Typing Rules

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

Definition 3 (Expression Type Correctness $T \vdash \mathtt{E}$)
An expression E *is* type correct *in an environment T, noted $T \vdash \mathtt{E}$, if there exist z and z' such that $T \vdash \mathtt{E} : (z, z')$ with wff(z) and wff(z').*

The static semantics inference rules are defined in Table 1; 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. In Rule (:), 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 (,), 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 $\mathtt{E}_1$ to the input of $\mathtt{E}_2$ is possible.

The most involved rule deals with loops ($\sim$). 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 loopedback 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 4 (Impedance Widening $\widehat{z}$)
The widened impedance *of z, noted $\widehat{z}$, is such that $|\widehat{z}| = |z|$ and $\forall i \in [1, |z|].\widehat{z}[i] = \widehat{z[i]}$, with:*

- $\widehat{\mathrm{vector}_n(t)^f} = \mathrm{vector}_n(\widehat{t})^f$;
- $\widehat{b[a]^f} = b[\widehat{a}]^f$;
- $\widehat{[n, m]} = [-\omega, +\omega]$.

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

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

6. DYNAMIC SEMANTICS

Since Faust sees parallelism as an implementation issue, 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 events. 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 :

$$v \in \mathrm{Val} \quad = \quad \mathbb{N} + \mathbb{R} + \bigcup_{n \geq 0} \mathrm{Val}^n + \{\bot\} + \{?\}$$

Since the evaluation process may be non-terminating, we posit that Val is a cpo, with bottom element $\bot$; 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 $v \in \mathrm{Val}^n$, represented by tuples of n elements, we define its size $|v|$ by $v \in \mathrm{Val}^{|v|}$.

A signal s, which is a history denoted by a function, is a member of Signal = $\mathrm{Time} \rightarrow \mathrm{Val}$. 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. We gather signals into beams $m = (m_1, ..., m_n)$ in $\mathrm{Beam} = \bigcup_{n \geq 0} \mathrm{Signal}^n$.

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:

$$\begin{aligned} r_0(\_) &= \lambda(s).(s) \\ r_0(+) &= \lambda(s_1, s_2).(\lambda t.s_1(t) + s_2(t)) \\ r_0(\mathtt{mem}) &= \lambda(s).(\lambda t.s(t-1) \text{ if } t \geq 1, 0 \text{ if } t = 0) \end{aligned}$$

These definitions assume that $T \vdash \mathtt{0} : t$ for all types t, since this is needed for the definition of mem to make sense.

As in the static semantics, introducing the vector extension in the dynamic semantics [4] simply amounts to adding, beside the value $\lambda().(\lambda t.())$ for {}, four straightforward bindings in the initial state:

- $r_0(\mathtt{vectorize}) =$
  $\lambda(s_1, s_2).(\lambda t.(s_1(nt), ..., s_1(n-1+nt)),$
  $\text{where } n = s_2(0))$;
- $r_0(\#) = \lambda(s_1, s_2).(\lambda t.s_1(t) \| s_2(t))$;
- $r_0(\mathtt{[]}) = \lambda(s_1, s_2).(\lambda t.s_1(t)[s_2(t)])$;
- $r_0(\mathtt{serialize}) =$
  $\lambda(s).(\lambda t.\ \bot, \text{ if } n = |s(0)| = 0,$
  $s(\lfloor t/n \rfloor)[\mathrm{mod}(t, n)] \text{ otherwise}).$

To be able to properly define the semantic function E:

$$E \in \mathrm{Exp} \rightarrow \mathrm{State} \rightarrow \mathrm{Beam} \rightarrow \mathrm{Beam}$$

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

Definition 5 (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 $T \vdash \mathtt{I}$.*

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

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

$$\text{(:)}\ \frac{T \vdash \mathtt{E}_1 : (z_1,z_1') \quad T \vdash \mathtt{E}_2 : (z_1',z_2')}{T \vdash \mathtt{E}_1 : \mathtt{E}_2 : (z_1,z_2')}$$

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

$$\text{(,)}\ \frac{T \vdash \mathtt{E}_1 : (z_1,z_1') \quad T \vdash \mathtt{E}_2 : (z_2,z_2')}{T \vdash \mathtt{E}_1,\mathtt{E}_2 : (z_1\|z_2, z_1'\|z_2')}$$

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

$$\text{(}\subset\text{)}\ \frac{T \vdash \mathtt{E} : (z,z') \quad z' \subset z_1' \quad z_1 \subset z}{T \vdash \mathtt{E} : (z_1,z_1')}$$

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

Table 1. Faust Static Semantics

$$\begin{aligned}
E[\![\mathtt{I}]\!]r &= r(\mathtt{I}) \\
E[\![\mathtt{E}_1 : \mathtt{E}_2]\!]r &= p_2 \circ p_1 \\
E[\![\mathtt{E}_1, \mathtt{E}_2]\!]r &= \lambda m.p_1(m[1,d_1]) \| p_2(m[d_1+1, d_1+d_2]) \\
E[\![\mathtt{E}_1 <: \mathtt{E}_2]\!]r &= \lambda m.p_2(\|_{1,d_2,d_1'} \lambda i.p_1(m)) \\
E[\![\mathtt{E}_1 :> \mathtt{E}_2]\!]r &= \lambda m.p_2(\|_{1,d_2,1} \lambda i.\mathrm{sum}(p_1(m)[i,d_1',d_2])) \\
& \quad \text{where } \mathrm{sum}((s)) = (s) \text{ and } \mathrm{sum}((s)\|m) = r(+)((s)\|\mathrm{sum}(m)) \\
E[\![\mathtt{E}_1 \sim \mathtt{E}_2]\!]r &= \lambda m.fix(\lambda m'.p_1(p_2(@(m'[1,d_2]))\|m)) \\
& \quad \text{where } @(()) = () \text{ and } @((s)\|m) = E[\![\mathtt{mem}]\!]rs\|@(m)
\end{aligned}$$

Table 2. Faust Denotational Semantics: we note $p_i = E[\![\mathtt{E}_i]\!]r$ and $(d_i, d_i') = dim(p_i)$

The semantics $E[\![\mathtt{E}]\!]r$ of an expression E in a type-correct state r is a function that maps an input beam m to an output beam m'.

The semantics (see Table 2) 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 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 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 $\mathrm{mod}(i, d_2)$-th output values of p_1 are summed together to construct the i-th input value of p_2. The loop expression 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 version of m'.

6.3 Frequency Correctness Theorem

In the presence of signals using different rates at run time, the consistency of their frequency 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 frequencies; this is the Frequency Correctness theorem. Of course, this theorem is only valid if the values denoted by a given Faust expression are consistent with its type definition, and kept as such all along execution (see the Subject Reduction theorem linking Faust static and dynamic semantics in [11]). We proceed first with the definition of this notion of runtime type correctness.

Definition 6 (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 $n \le v \le m$;*
- *when $v \in \mathbb{R}$, then $t = \mathrm{float}[n,m]$ and $n \le v \le m$;*
- *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 7 (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 8 (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 introduce consistent definitions for their domains:

Definition 9 (State Type Consistency $\vdash T, r$)
An environment T and a state r are consistent*, noted $\vdash T, r$, if, for all* `I` *in $dom(r)$, for all z, z', m, one has: if $T \vdash$ `I` $: (z, z')$ and $m : z$, then $r(\texttt{I})(m) : z'$ and $dim(r(\texttt{I})) = (|z|, |z'|)$.*

We may now proceed with the issue of frequency.

Definition 10 (Beam Boundness (m, z) ! c)
*For any $c \in \mathbb{Q}$, a beam 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 at most $c\sharp(z[i^*])$ elements in its domain of definition [5] . 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 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 frequency, its unit is the second, and thus c is a time. The definition of Beam 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 states that, given a Faust expression `E` (with no explicit `mem`, since its delaying action extends domains of definition), 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 c-bounded, then the semantics $p(m)$ of `E` will yield a c-bounded beam m' of impedance z'.

Theorem 1 (Frequency Correctness)
For all `E` *not containing* `mem`*, T, z, z', c, r, m and m', if $\vdash T, r,\ m : z,\ (m, z)$! c, $T \vdash$ `E` $: (z, z')$, then $|z'| = 0 \vee (m', z')$! c, where $m' = p(m) : z'$ and $p = E[\![\texttt{E}]\!]r$.*

Basically, this theorem (see proof in [11]) 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`. This shows that the synchronous nature of Faust is preserved.

[5] 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.

7. CONCLUSION

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

8. ACKNOWLEDGEMENTS

This work is partially funded by the French ANR, as part of the ASTREE Project (2008 CORD 003 01).

9. REFERENCES

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

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

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

[4] E. Gaudrain and 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, and S. Letz, "Syntactical and Semantical Aspects of Faust," *Soft Computing*, vol. 8, no. 9, pp. 623–632, 2004.

[7] Y. Orlarey, "Notes sur les extensions de Faust," tech. rep., GRAME, Lyon, 2009.

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

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

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

[11] P. Jouvelot and Y. Orlarey, "Semantics for Multirate Faust," tech. rep., CRI, Mathématiques et systèmes, MINES ParisTech, Nov. 2009.