Syntactical and Semantical Aspects of Faust

Yann Orlarey, Dominique Fober, Stéphane Letz

Soft Computing manuscript No.
(will be inserted by the editor)

Syntactical and Semantical Aspects of Faust

Yann Orlarey, Dominique Fober, Stephane Letz

Grame, Centre National de Creation Musciale, Lyon, France

Received: date / Revised version: date

Abstract This paper presents some syntactical and semantical aspects of FAUST (Functional AUdio STreams), a programming language for real-time sound processing and synthesis. The programming model of FAUST combines two approaches : *functional programming* and *block-diagrams composition*. It is based on a *block-diagram algebra*. It has a well defined formal semantic and can be compiled into efficient C/C++ code.

Key words functional programming – real-time – signal processing – dataflow – compiler

1 Introduction

FAUST (Functional AUdio STreams), is a programming language for real-time signal processing and synthesis. It targets high-performance signal processing applications and audio plugins. It has been designed with three main goals in mind : expressiveness, clean mathematical semantics and efficiency.

Expressiveness is achieved by combining two approaches: functional programming and algebraic block-diagrams (extended function composition). This computation model has also the advantage of a simple and well defined formal semantics.

Having clean semantics is not just of academic interest. It allows the Faust compiler to be *semantically driven*. Instead of compiling the block-diagram itself, it compiles "what the block-diagram compute". It also allows to discover simplifications and factorizations to produce efficient code.

Faust is build on top of a *block-diagram algebra* that we will describe in the next section. This algebra is totally independant of Faust and could be reused in a completely different domain. This is why in section 2 we will first present it without any reference to Faust and its signal processing semantic.

It is only in section 3 that we will relate the *block-diagram algebra* to Faust and to the signal processing semantic. We will also describe the complete set of primitives of the language.

In section 4 we will give a concrete example of usage of Faust with the Karplus-Strong algorithm. In section 5 we will give an overview of how the Faust compiler works. We will end the paper with some concluding remarks and future directions of work.

2 The block-diagram algebra

Block-diagram formalisms are widely used in visual languages particularly in musical languages. The user creates programs (i.e. block-diagrams), by connecting graphical *blocks* which represent the functionalities of the system. In almost every implementation, a block-diagram is represented internally as a graph, and interpreted as a *dataflow* computation (see [2] and[1] for historical papers on dataflow, and [7] or [4] for surveys).

This very common approach has several drawbacks:

1. Due to their generality, the semantics of dataflow models can be quite complex. It depends on many technical choices like for example, synchronous or asynchronous computations, deterministic or non-deterministic behavior, bounded or unbounded communication FIFOs, firing rules, etc. Because of this complexity, the vast majority of dataflow inspired music languages have no *explicit* formal semantic. The semantics is hidden in the dataflow engine. The real behavior of a block-diagram can be difficult to understand without a good knowledge of the implementation.
2. Dataflow models are difficult to implement efficiently and most of the time no compiler exists, only an interpreter is provided. In order to minimize interpretation overheads, computations typically operate on block of samples instead of individual samples. This comes with a cost : recursive computations are nearly impossible to implement and therefore many common signal processing operations can't be implemented and must be provided as primitives or external plug-ins.

3. Graphs are complex to manipulate. For example it might be desirable to algorithmically generate block-diagrams using templates and macros. But this is very difficult if the block diagram is represented by a graph. Moreover, it can be very useful to provide, in addition to the visual syntax, a textual syntax that can be edited with a simple text editor and processed with standard tools.

As we will see in the following paragraphs, these problems can be solved giving up the *graph representation* and adopting an equivalent *tree representation* based on a small algebra of *composition operators* [5].

This *block-diagram algebra* (BDA) is the heart of Faust. But it is independent from it. Therefore in this section we will focus on the *topological semantic* of the BDA: how things are connected, but not what they do. We will assume the existence of a set of primitive building blocks, but without further details or any reference to the signal processing semantic of Faust.

2.1 Definitions

Let $\mathbb{B}$ be a set of primitive blocks (this set is leaved undefined at this stage) and let $\mathbb{D}$ be the set of block-diagrams build on top of $\mathbb{B}$. A block-diagram $D \in \mathbb{D}$ is either an *identity* block ($\_$), a *cut* block ($!$), a primitive block $B \in \mathbb{B}$, or composition of two block-diagrams based on one of the five composition operator of the algebra. More formally a block-diagram $D \in \mathbb{D}$ is defined recursively by the following syntactic rule :

$$D = \_ \mid ! \mid B \mid (D_1 : D_2) \mid (D_1, D_2) \\ \mid (D_1 <: D_2) \mid (D_1 :> D_2) \mid (D_1 \sim D_2)$$

We will adopt a type-like notation : $D : i_D \rightarrow o_D$ to indicate that block-diagram $D \in \mathbb{D}$ has i_D inputs and o_D outputs thus notated :

$$\text{inputs}(D) = \{D_{in}[0], D_{in}[1], \ldots, D_{in}[i_D - 1]\}$$

$$\text{outputs}(D) = \{D_{out}[0], D_{out}[1], \ldots, D_{out}[o_D - 1]\}$$

2.2 Sequential composition $(A : B)$

The *sequential composition* operator is used to connect the outputs of A to the corresponding inputs of B (such that $A_{out}[i]$ is connected to $B_{in}[i]$). The inputs of $(A : B)$ are the inputs of A and the outputs of $(A : B)$ are the outputs of B.

If the number of inputs and outputs are not the same, the exceeding outputs of A (resp. the exceeding inputs of B) form additional outputs (resp. inputs) of the resulting block-diagram (see figure 1). The number of inputs and outputs of the resulting block-diagram is given by the following three rules :

$$\frac{A : i_A \rightarrow o_A \qquad B : i_B \rightarrow o_B \qquad o_A = i_B}{(A : B) : i_A \rightarrow o_B}$$

$$\frac{A : i_A \rightarrow o_A \qquad B : i_B \rightarrow o_B \qquad o_A > i_B}{(A : B) : i_A \rightarrow o_B + o_A - i_B}$$

$$\frac{A : i_A \rightarrow o_A \qquad B : i_B \rightarrow o_B \qquad o_A < i_B}{(A : B) : i_A + i_B - o_A \rightarrow o_B}$$

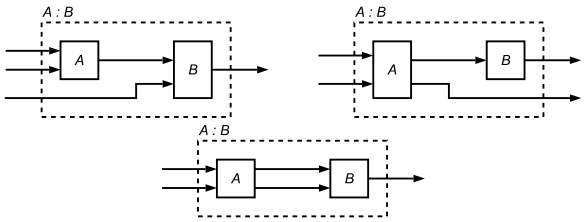

Fig. 1 The sequential composition operator : possible cases according to the number of outputs of A and inputs of B

2.3 Parallel composition (A, B)

The *parallel composition* operator associates two block-diagrams one on top of the other, without connections. The inputs (resp. the outputs) of (A, B) are the inputs (resp. the outputs) of A and B as defined in the following rule :

$$\frac{A : i_A \rightarrow o_A \qquad B : i_B \rightarrow o_B}{(A, B) : i_A + i_B \rightarrow o_A + o_B}$$

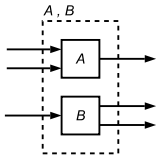

Fig. 2 The parallel composition operator

2.4 Split composition $(A <: B)$

This *split composition* operator is used to distribute the outputs of A to several inputs of B. It requires that the number of inputs of B is an exact multiple of the number of outputs of A. For example if A has 3 outputs and B has 6 inputs, then each output of A will be connected to 2 inputs of B. The general rule is that, if $A : i_A \rightarrow m$ and $B : o_A * k \rightarrow q$ then $A_{out}[i]$ is connected to $B_{in}[i + j * o_A]$ where $j < k$. The inputs (resp. the outputs) of $(A <: B)$ are the inputs of A (resp. the outputs of B) as defined in the following rule :

$$\frac{A : i_A \rightarrow o_A \qquad B : o_A * k \rightarrow o_B}{(A <: B) : i_A \rightarrow o_B}$$

Because we suppose that the number of inputs of B is an exact multiple of the number of outputs of A, the split composition is a partial function over $\mathbb{D}$. It would have been easy to extend the semantic of $<:$ to cover all possible cases, but experiments with Faust have proved that these restrictions are useful to discover potential programming errors. The same remark applies to the restrictions introduced in the $:>$ and $\sim$ operators presented in the next paragraphs. Violations of these restrictions are typically flagged by the compiler and reported as typing errors.

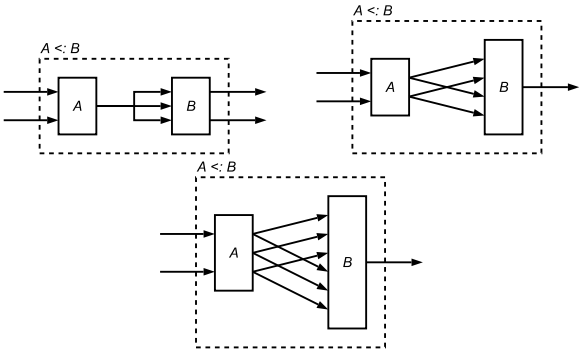

Fig. 3 The split composition operator

Note that if $k = 1$, then $A <: B$ is equivalent to $A : B$.

2.5 Merge composition $(A :> B)$

As suggested by the notation, the *merge composition* operator does the inverse of the split operator. It is used to connect several outputs of A to the same inputs of B. It requires that the number of outputs of A be an exact multiple of the number of inputs of B. The general rule is that, if $A : i_A \to i_B * k$ and $B : i_B \to o_B$ then $A_{out}[i + j * i_B]$ is connected to $B_{in}[i]$ where $j < k$.

The inputs (resp. the outputs) of $(A :> B)$ are the inputs of A (resp. the outputs of B). The number of outputs of A should be an exact multiple of the number of inputs of B:

$$\frac{A : i_A \to i_B * k \qquad B : i_B \to o_B}{(A :> B) : i_A \to o_B}$$

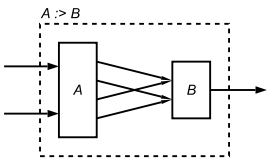

Fig. 4 The merge composition operator

Note that if $k = 1$, then $(A :> B)$ is equivalent to $(A : B)$

.

2.6 Recursive composition $(A \sim B)$

The *recursive composition* is used to create cycles in the block-diagram in order to express recursive computations. Each input of B is connected to the corresponding output of A. Each output of B is connected to the corresponding input of A.

The inputs of $(A \sim B)$ are the remaining inputs of A. The outputs of $(A \sim B)$ are the outputs of A. Two examples of recursive composition are given in figure 5.

$$\frac{A : i_A \to o_A \qquad B : i_B \to o_B \qquad o_B \leq i_A \qquad i_B \leq o_A}{(A \sim B) : i_A - o_B \to o_A}$$

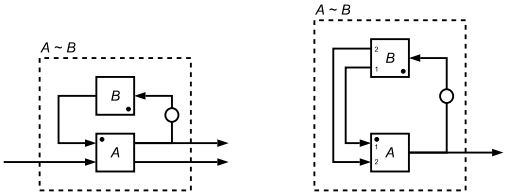

Fig. 5 Two examples of *recursive composition*

2.7 Identity block _ and Cut block !

For the algebra to be complete we need to introduce two additional elements : the *identity block* (a simple connection wire) represented by the underscore symbol _ and the *cut block* (a connection ending) represented by the symbol !. These two elements are represented figure 6. We have :

$$\frac{}{\_ : 1 \to 1}$$

and

$$\frac{}{! : 1 \to 0}$$

The *identity block* and *cut block* are typically used to create complex routings. For example to cross two connections, that is to connect $A_{out}[0]$ to $B_{in}[1]$ and $A_{out}[1]$ to $B_{in}[0]$ one can write :

$$A : (\_, \_ <:!, \_, \_, !) : B$$

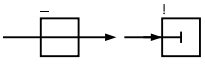

Fig. 6 The *identity* and *cut* primitive boxes

2.8 Precedence and associativity

In order to simplify the expressions and to avoid too many parenthesis, we define a precedence and an associativity for each operator as given in the following table :

Priority	Symbol	Name	Associativity
3	~	recursive	Left
2	,	parallel	Right
1	:, <:, :>	sequential, split, merge	Right

Based on these rules we can write :

$$a : b, c \sim d, e : f$$

instead of

$$(a : (((b, (c \sim d)), e) : f))$$

Moreover, the following properties hold :

$$((A : B) : C) = (A : (B : C))$$

$$((A, B), C) = (A, (B, C))$$

$$((A <: B) <: C) = (A <: (B <: C))$$

$$((A :> B) :> C) = (A :> (B :> C))$$

2.9 Stefanescu Algebra of Flownomials

Our block-diagram algebra is related to Gh. Stefanescu [6] *Algebra of Flownomials* (AoF) proposed to represent directed flowgraphs (blocks diagrams in general including flowcharts) and their behaviors.

The AoF is presented as an extension of Kleene's calculus of regular expressions. It is based on three operations and various constants used to describe the branching structure of the flowgraphs. They all have a direct translation into our BDA as shown table 1.

Although the AoF and the BDA are equivalent in that they can both represent any block-diagram, the AoF lacks the high-level composition operations offered by the BDA and it is less suited for a practical programming language.

AoF		BDA
par. comp.	$A+B$	A, B
seq. comp	$A.B$	$A : B$
feedback	$A\uparrow$	$A \sim \_ :!$
identity	I	$\_$
transposition	X	$\_, \_ <:!, \_, \_, !$
ramification	$\wedge_k^n$	$(\_, \ldots)_n <: (\_, \ldots)_{n*k}$
	$\wedge_0$	!
identification	$\vee_n^k$	$(\_, \ldots)_{n*k} :> (\_, \ldots)_n$

Table 1 Correspondences between the algebra of Flownomials and the block diagram algebra. Note : $(\_, \ldots)_n$ means the composition of n identity in parallel.

3 The Faust language

The Faust language is built on top of the BDA, extended with a suitable set of primitives and some additional syntactic constructions allowing to define a Faust program as a list of definitions.

This section will start with some few definitions related to the semantic of signal processor. Then the signal processing semantic of the BDA will be presented. The section will end with a description of Faust primitives.

3.1 Notations and definitions

A Faust block-diagram denotes a *signal processor* transforming input signals into output signals. In this paragraph we define the notions of *signal*, of *signal processor* and some notations.

3.1.1 Signals A *signal* s is a discrete function of time $s : \mathbb{N} \to \mathbb{R}$. The value of signal s at time t is written $s(t)$. By convention, the full range of the AD/DA converters corresponds to samples values between -1.0 and $+1.0$. We denote by $\mathbb{S}$ to be the set of all possible signals : $\mathbb{S} = \mathbb{N} \to \mathbb{R}$.

3.1.2 Constant Signals A signal is a *constant signal* if it always delivers the same value : $\exists v \in \mathbb{R}, \forall t, s(t) = v$. We notate $\mathbb{S}_k \subset \mathbb{S}$ the subset of *constant signals*.

3.1.3 Integer Signals A signal is an *integer signal* if it always delivers integer values : $\forall t, s(t) \in \mathbb{Z}$. We notate $\mathbb{S}_i \subset \mathbb{S}$ the subset of *integer signals*. Moreover we have $\mathbb{S}_i = \mathbb{N} \to \mathbb{Z}$.

3.1.4 Constant Integer Signals A signal is an *constant integer signal* if it always delivers the same integer value : $\exists k \in \mathbb{Z}, \forall t, s(t) = k$. We notate $\mathbb{S}_{ik} \subset \mathbb{S}$ the subset of *constant integer signals*. We have $\mathbb{S}_{ik} = \mathbb{S}_i \cap \mathbb{S}_k$.

3.1.5 Tuples of Signals Signal processors operate on *tuples of signals*. We will write $(x_1, \ldots, x_n)$: a *n*-tuple of signals element of $\mathbb{S}^n$. The *empty tuple*, single element of $\mathbb{S}^0$ is notated $()$.

3.1.6 Signal processors Faust primitives and block-diagrams represent *signal processors*, functions transforming *input signals* to produce *output signals*. A *signal processors* p is a function from *n*-tuples of signals to *m*-tuples of signals $p : \mathbb{S}^n \to \mathbb{S}^m$. We notate $\mathbb{P}$ the set of all signal processors :

$$\mathbb{P} = \bigcup_{n,m} \mathbb{S}^n \to \mathbb{S}^m$$

3.1.7 Semantic function In order to explicitly refer to the mathematical *meaning* of a block-diagram D and to distinguish it from its syntactic representation we will used *semantic brackets* : $[\![\ ]\!]$. The notation $[\![\, D \,]\!]$ means : *the signal processor represented by block-diagram* D. Therefore $[\![\, . \,]\!]$ as a *semantic function* translating *block-diagrams* into *signal processors* : $[\![\, . \,]\!] : \mathbb{D} \rightarrow \mathbb{P}$.

3.2 Semantic of the Block-Diagram Algebra

In the previous section we have informally presented the *topological semantic* of the Block-Diagram Algebra, how things are connected, but without any references to the actual meaning of these connections. In this paragraph we will define the *signal processing semantic* of the various composition operators.

3.2.1 Sequential composition The result of the sequential composition of $D_1 : n \rightarrow m$ and $D_2 : p \rightarrow q$ is defined by different rules according to m and p :

$$\frac{\begin{array}{l}[\![\, D_1 \,]\!](x_1, \ldots, x_n) = (s_1, \ldots, s_m) \\ [\![\, D_2 \,]\!](s_1, \ldots, s_p) = (y_1, \ldots, y_q) \\ (m = p)\end{array}}{[\![\, D_1 : D_2 \,]\!](x_1, \ldots, x_n) = (y_1, \ldots, y_q)}$$

$$\frac{\begin{array}{l}[\![\, D_1 \,]\!](x_1, \ldots, x_n) = (s_1, \ldots, s_m) \\ [\![\, D_2 \,]\!](s_1, \ldots, s_m, z_1, \ldots, z_{p-m}) = (y_1, \ldots, y_q) \\ (m < p)\end{array}}{[\![\, D_1 : D_2 \,]\!](x_1, \ldots, x_n, z_1, \ldots, z_{p-m}) = (y_1, \ldots, y_q)}$$

$$\frac{\begin{array}{l}[\![\, D_1 \,]\!](x_1, \ldots, x_n) = (s_1, \ldots, s_p, s_{p+1}, \ldots, s_m) \\ [\![\, D_2 \,]\!](s_1, \ldots, s_p) = (y_1, \ldots, y_q) \\ (m > p)\end{array}}{[\![\, D_1 : D_2 \,]\!](x_1, \ldots, x_n) = (y_1, \ldots, y_q, s_{p+1}, \ldots, s_m)}$$

It is easy to deduce from the above rules that sequential composition is an associative operation : $(D_1 : D_2) : D_3 = D_1 : (D_2 : D_3)$

3.2.2 Parallel composition The result of the parallel composition of $D_1 : n \rightarrow m$ and $D_2 : o \rightarrow p$ is defined by :

$$\frac{\begin{array}{c}[\![D_1]\!](x_1, \ldots, x_n) = (y_1, \ldots, y_m) \\ [\![D_2]\!](s_1, \ldots, s_o) = (t_1, \ldots, t_p)\end{array}}{[\![D_1, D_2]\!](x_1, \ldots, x_n, s_1, \ldots, s_o) = (y_1, \ldots, y_m, t_1, \ldots, t_p)}$$

The associativity holds also for the parallel composition : $(D_1, D_2), D_3 = D_1, (D_2, D_3)$

3.2.3 Split composition In the split composition, the output signals of $D_1 : n \rightarrow m$ are duplicated k times and distributed to the inputs of $D_2 : m.k \rightarrow p$:

$$\frac{\begin{array}{c}[\![D_1]\!](x_1, \ldots\ x_n) = (s_1, \ldots\ s_m) \\ [\![D_2]\!](\overbrace{s_1, \ldots\ s_m}^{1}, \ldots, \overbrace{s_1, \ldots\ s_m}^{k}) = (y_1, \ldots\ y_p)\end{array}}{[\![D_1 <: D_2]\!](x_1, \ldots\ x_n) = (y_1, \ldots\ y_p)}$$

3.2.4 Merge composition In the merge composition, the output signals of $D_1 : n \rightarrow m.k$ are added together by groups of k signals and sent to the corresponding input of $D_2 : m \rightarrow p$:

$$\frac{\begin{array}{l}[\![D_1]\!](x_1, \ldots\ x_n) = (s_1, \ldots\ s_{m.k}) \\ [\![D_2]\!](\sum_{j=0}^{k-1}(s_{1+j.m}), \ldots\ \sum_{j=0}^{k-1}(s_{m+j.m})) = (y_1, \ldots, y_p)\end{array}}{[\![D_1 :> D_2]\!](x_1, \ldots\ x_n) = (y_1, \ldots\ y_p)}$$

3.2.5 Recursive composition In the recursive composition of $D_1 : v + n \rightarrow u + m$ and $D_2 : u \rightarrow v$ the first u output signals of D_1 are sent with a 1-sample delay to the corresponding inputs of D_2. The outputs of D_2 are sent to the first v inputs of D_1 :

$$\frac{\begin{array}{l}[\![D_1]\!](r_1, \ldots, r_v, x_1, \ldots, x_n) = (y_1, \ldots, y_m) \\ [\![D_2]\!](y_1^{-1}, \ldots, y_{u \leq m}^{-1}) = (r_1, \ldots, r_v)\end{array}}{[\![(D_1 \sim D_2)]\!](x_1, \ldots, x_n) = (y_1, \ldots, y_m)}$$

For a signal x, the notation x^{-1}, represents the signal x delayed by one sample such that : $\forall x \in \mathbb{S}\ x^{-1}(0) = 0$ and $x^{-1}(t+1) = x(t)$. The resulting tuple of signals $(y_1, \ldots, y_m)$ is the least fixed point that satisfies the equation. This fixed point always exists as we limit ourselves to recursive computation depending only of past values.

3.2.6 Identity box _ and Cut box ! As shown in figure 6, the *identity* primitive $(\_)$ is essentially a simple wire representing the identity function for signals :

$$[\![\ \_\ ]\!] : \mathbb{S} \rightarrow \mathbb{S}$$
$$[\![\ \_\ ]\!](s) = (s)$$

The *cut* box with one input and no output is used to end a connection :

$$[\![\, ! \,]\!] : \mathbb{S} \rightarrow \mathbb{S}^0$$
$$[\![\, ! \,]\!](s) = ()$$

3.3 Faust Primitives

The set $\mathbb{B}$ of Faust primitives follows as much as possible the set of C/C++ operators. In order to guarantee the role of signal processing specification language of Faust, typical signal processing operations are not part of the primitives. They are typically implemented in Faust and provided as external libraries.

3.3.1 Arithmetic primitives Faust arithmetic primitives correspond to the five C/C++ operators $+ - \times / \%$ represented figure 7. The semantics scheme of each of these primitives is the same. For an operation $\star \in \{+, -, \times, /, \%\}$ we have :

$$\begin{aligned} [\![\star]\!] &: \mathbb{S}^2 \to \mathbb{S} \\ [\![\star]\!](s_1, s_2) &= (y) \\ y(t) &= s_1(t) \star s_2(t) \end{aligned}$$

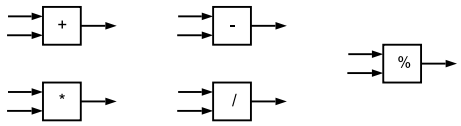

Fig. 7 The arithmetic primitives

3.3.2 Comparison primitives The six comparison primitives are also available : $<, >, <=, >=, !=, ==$. They compare two signals and produce a boolean signal. For a comparison $\bowtie \in \{<, >, <=, >=, !=, ==\}$ we have :

$$\begin{aligned} [\![\bowtie]\!] &: \mathbb{S}^2 \to \mathbb{S} \\ [\![\bowtie]\!](s_1, s_2) &= (y) \\ y(t) &= \begin{cases} 1 \text{ if } s_1(t) \bowtie s_2(t) \\ 0 \text{ else} \end{cases} \end{aligned}$$

3.3.3 Bitwise primitives Bitwise primitives corresponding to the five C/C++ operators $<<, >>, \&, |, \wedge$ are also provided. Again the semantics scheme of each of these primitives is the same. For an operation $\star \in \{<<, >>, \&, |, \wedge\}$ we have :

$$\begin{aligned} [\![\star]\!] &: \mathbb{S}^2 \to \mathbb{S} \\ [\![\star]\!](s_1, s_2) &= (y) \\ y(t) &= s_1(t) \star s_2(t) \end{aligned}$$

3.3.4 Constants Constants are represented by boxes with no input and a constant output signal (see figure 8). For a number $k \in \mathbb{R}$ we have

$$\begin{aligned} [\![k]\!] &: \mathbb{S}^0 \to \mathbb{S} \\ [\![k]\!]() &= (y) \\ y(t) &= k \end{aligned}$$

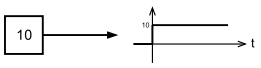

Fig. 8 The constant 10

3.3.5 Casting Two primitives : `float` and `int` are provided to cast signals to floats or integers.

Fig. 9 The *int cast* and *float cast* primitive boxes

3.3.6 Foreign definitions Foreign definitions are used to incorporate externally defined C functions and constants. Foreign functions are declared using the reserved keyword `ffunction`, specifying the C prototype, the include file, and the library to link against.

`ffunction(` *prototype* `,` *include* `,` *library* `)`

For example the *sin* function is declared :

```
ffunction( float sin(float), <math.h>, "-lm")
```

Foreign constants are declared using the reserved word `fconstant` :

```
fconstant(int fSamplingFreq, <math.h>)
```

3.3.7 Fixed delays Fixed delays are provided with two primitives `@` and `mem`. More sophisticated delays are implemented using the read-write tables. While `@` represent a fixed delay :

$$\begin{aligned} [\![@]\!] &: \mathbb{S} \times \mathbb{S}_{ik} \to \mathbb{S} \\ [\![@]\!](x, d) &= (y) \\ y(t + d(t)) &= x(t) \end{aligned}$$

The `mem` represent a 1-sample delay :

$$\begin{aligned} [\![\text{mem}]\!] &: \mathbb{S} \to \mathbb{S} \\ [\![\text{mem}]\!](x) &= (y) \\ y(t + 1) &= x(t) \end{aligned}$$

We have $\forall s, [\![\text{mem}]\!](x) = [\![@]\!](x, 1)$.

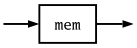

Fig. 10 The *mem* box represents a one sample delay

3.3.8 Read-only table The read-only table `rdtable` is a primitive box with 3 inputs : a constant size signal, an initialization signal and an index signal. It produce an output signal by reading the content of the table.

$$\begin{aligned} [\![\text{rdtable}]\!] &: \mathbb{S}_{ik} \times \mathbb{S} \times \mathbb{S}_i \to \mathbb{S} \\ [\![\text{rdtable}]\!](n, v, i) &= (y) \\ y(t) &= v(i(t)) \end{aligned}$$

The size of the table is determined by the constant signal n. The index signal i is such that $\forall t, 0 \leq i(t) < n$.

3.3.9 Read-write table The read-write table `rwtable` is almost the same as the `rdtable` box, except that the data stored at initialization time can be modified. It has 2 more inputs streams : the write index and the write signal.

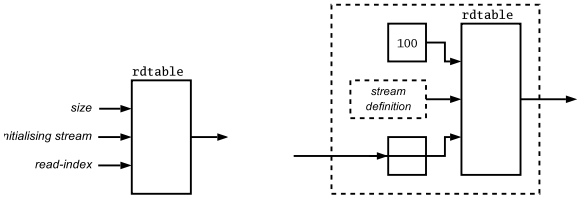

Fig. 11 The read-only table primitive

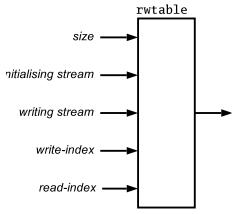

Fig. 12 The read-write table primitive

3.3.10 Selectors The primitives `select2` and `select3` allow to dynamically select between 2 or 3 signals according to a selector signal. The `select2` box receives 3 input streams, the selection signal, and the two signals.

$$\begin{aligned} [\![\text{select2}]\!] &: \mathbb{S}_i \times \mathbb{S}^2 \to \mathbb{S} \\ [\![\text{select2}]\!](i, s[0], s[1]) &= (y) \\ y(t) &= s[i(t)](t) \end{aligned}$$

The index signal i is such that $\forall t, i(t) \in \{O, 1\}$. The `select3` box is exactly the same except that it selects between 3 signals :

$$\begin{aligned} [\![\text{select3}]\!] &: \mathbb{S}_i \times \mathbb{S}^3 \to \mathbb{S} \\ [\![\text{select3}]\!](i, s[0], s[1], s[2]) &= (y) \\ y(t) &= s[i(t)](t) \end{aligned}$$

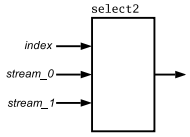

Fig. 13 The *select2* primitive box

3.3.11 Graphic user interface A Faust block-diagram can contain user interface elements (buttons, sliders, etc.) grouped together according to different layout strategies. Like every thing in Faust, user interface elements deliver signals. It is therefore possible to mix user interface elements with other signal processing operations.

The `button` primitive has the following syntax:

`button("`*label*`")`

The signal delivered by the button reflects the user actions:

$$\begin{aligned} [\![\text{button}(\text{"}label\text{"})]\!] &: \mathbb{S}^0 \to \mathbb{S} \\ [\![\text{button}(\text{"}label\text{"})]\!]() &= (y) \\ y(t) &= \begin{cases} 1 \text{ when the button is pressed} \\ 0 \text{ otherwise} \end{cases} \end{aligned}$$

This box is a monostable trigger. It has no input, and one output that is set to 1 if the user click the button, and else to 0.

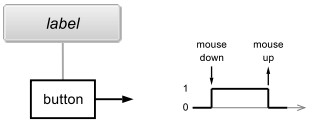

Fig. 14 The *button* primitive box

The `checkbox` is a bistable trigger. A mouse click sets the output to 1. A second mouse click sets the output back to 0.

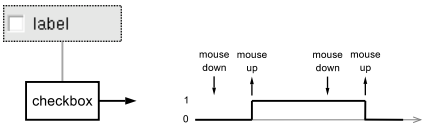

Fig. 15 The *checkbox* primitive box

Here's the syntax :

`checkbox("`*label*`")`

The slider boxes `hslider` (horizontal) and `vslider` (vertical) provide some powerful controls for the parameters. Here's the syntax :

`hslider("`*label*`",` *start*`,` *min*`,` *max*`,` *step*`)`

This produces a slider, horizontal or vertical, that let the user pick a value between *min* and *max*−*step*. The initial value is *start*. When the user moves the slider, the value changes by steps of the value of *step*. All the parameters can be `float`s or `int`s.

The associated box has no input, and one output which is the value that the slider displays.

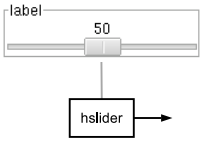

Fig. 16 The *slider* primitive box

This primitive displays a numeric entry field on the GUI. The output of this box is the value displayed in the field.

`nentry("`*label*`", `*start*`, `*min*`, `*max*`, `*step*`)`

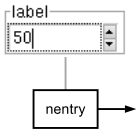

Fig. 17 The *nentry* primitive box

The layout of the user interface is controlled using *group expressions*. For example

`hgroup("`*label*`", `*D*`)`

defines an horizontal layout for all the user interface elements that appears in *D*. Similarly `vgroup("`*label*`", `*D*`)` defines a vertical layout and `tgroup("`*label*`", `*D*`)` a tabular organization.

4 Example : the Karplus-Strong Algorithm

Karplus-Strong is a well known algorithm first presented by Karplus and Strong in 1983 [3]. It can generate interesting metallic plucked-string and drum sounds. While non completely trivial, the principle of the algorithm is simple enough to be described in few lines of Faust.

An overview of the implementation is given figure 18. It uses an impulse of noise that goes into a resonator based on a delay line with feedback. The user interface contains a button to trig the sound and allows to control the size of both the resonator and the noise impulse, as well as the amount of feedback.

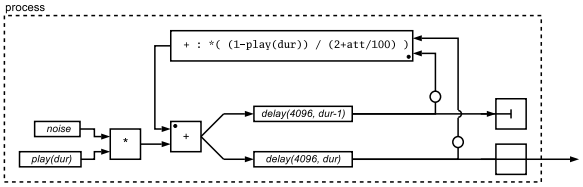

Fig. 18 The Faust implementation

4.1 The noise generator

The *noise* generator is based on a very simple *random* number generator which values are scaled down between -1 and $+1$. The following Faust definitions correspond to the block-diagram of figure 19.

```
random = (*(1103515245)+12345) ~ _;
noise = random *(1.0/2147483647.0);
```

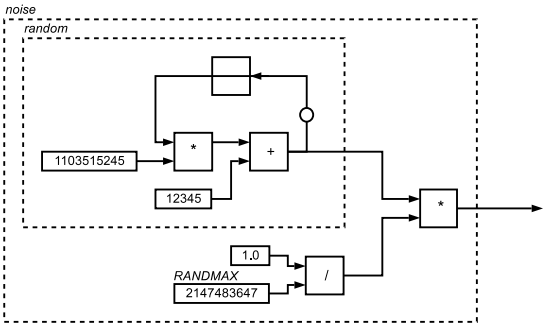

Fig. 19 The noise generator

4.2 The trigger

The trigger is used to deliver a one shot control signal every time the user press on the play button. The control signal must have a precise width that is independent of how long the user press on the button.

```
impulse(x) = (x - mem(x)) > 0;
release(n) = +~(_ <: -(>(0)/n)) : >(0);
trigger(n) = impulse : release(n);
```

Impulse (figure 20) transforms the play button signal into a one sample impulse. The a *release* is added to transform this impulse into a n-samples signal.

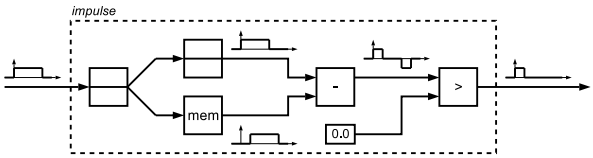

Fig. 20 `impulse` in charge of transforming a button signal into a one sample impulse

4.3 The resonator

The resonator uses a delay line implemented using a rwtable (see figure 21). It averages two consecutive samples with delay d and $d-1$ and feeds back the result with an attenuation a into the table.

```
index(n) = &(n-1)~+(1);
delay(n,d) = rwtable(n, 0.0, index(n), _,
(index(n)-int(d)) & (n-1)) ;
resonator(d,a) = (+ <: (delay(4096, d-1)
+ delay(4096, d))/2.0)~*(1.0-a) ;
```

4.4 Putting all together

The description in now almost complete. We can put all the pieces together with the user interface description and define

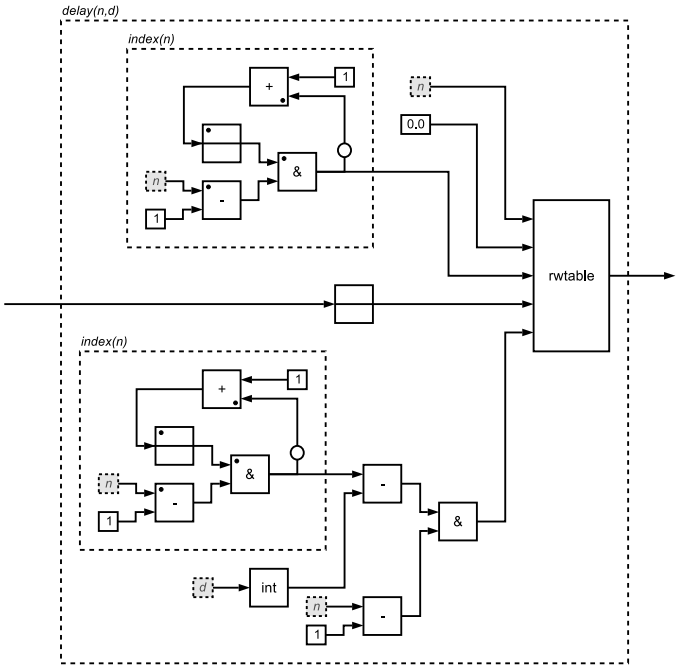

Fig. 21 The delay line

`process` that will produce the standalone application of figure 22.

```
dur = hslider("duration", 128, 2, 512, 1);
att = hslider("attenuation", 0.1, 0, 1, 0.01);

process = noise : *(button("play")
: trigger(dur)) : resonator(dur,att);
```

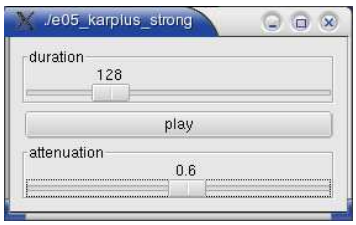

Fig. 22 The Karplus-Strong user interface automatically generated from the Faust specification

5 The Faust Compiler

The role of the Faust compiler is to translate a signal processor specification written in Faust into C/C++ code. Because we target high-performance real-time signal processing applications, the main challenge is to generate efficient code that can compete with hand-written one. The key idea is not to compile the block diagram itself, but *what it computes*. Driven by the semantic rules described in the previous sections, the compiler starts by propagating symbolic signals into the block diagram in order to discover how each output signal can be expressed as a function of the input signals. These resulting signal expressions are then simplified and common subexpressions are factorized. Finally they are translated into C/C++ code resulting in a five methods class that implements the specification.

5.1 The compilation process

The compilation process involves several phases that we describe briefly in the following paragraphs.

5.1.1 Parsing the source files As mentioned in the previous example, a faust program is an unordered list of definitions that includes a definition of the keyword `process`, the Faust equivalent to C `main()`. The first step is to parse all the input files in order to produce a internal dictionary of definitions. Each definition is represented as an abstract syntactic tree (AST). To simplify the discovery of common subtree, the AST are implemented using hash-consing such that syntactically equal trees are always shared in memory.

5.1.2 Evaluation of `process` The next step is to evaluate the definition of `process` stored in the dictionary. This step is basically a λ-calculus interpreter with a strict evaluation strategy. Names are replaced with their definition found in the dictionary and applications of abstractions are β-reduced. The result is "flat" block-block-diagram where everything have been expanded.

5.1.3 Type annotation of block-diagrams Using the rules described in section 2, every subtree D of the process tree is annotated with its number of inputs and outputs : $n \rightarrow m$.

5.1.4 Symbolic propagation In order to discover what the block-diagram computes, symbolic signals are propagated through it using the semantic rules described in section 3. This propagation results in a list of *signal expressions* expressing how each output signals is computed from the input signals.

5.1.5 Type annotation of signals The resulting signal expressions are type annotated according to several aspects :

1. the *nature* of the signal : *integer* of *float*.
2. the *computation time* of the signal: the signal can be computed at *compilation time*, at *initialization time* or at *execution time*.
3. the *speed* of the signal : *constant* signals computed only once, *low speed* user interface signals computed once for every block of samples, *high speed* audio signals. computed every samples.
4. parallelism of the signal : *true* is the samples of the signal can be computed in parallel, *false* otherwise.

5.1.6 Simplification and normalization The signal expressions are rearranged and simplified by executing all the computations that can be done at compilation time producing signal expressions in normal form.

5.1.7 Sharing of recursive signals The previous steps may have produced different, but α-equivalent, representations of recursive signals. Because they are syntactically different they are not automatically shared by the hash-consing technique. This step replaces all α-equivalent subtrees with a common shared subtree.

5.1.8 Reuse annotation The subtrees are annotated with a *reuse* flag indicating if the computation should be stored in a temporary variable to be later reused. The notion of reuse can be quite subtle. It concerns expressions that have several occurrences in the global expression (space reuse), but also expressions with only one occurrence but in a higher speed context (time reuse).

5.1.9 Code generation The compiler generates a C++ class that implements the Faust specification. This class may be wrapped into an *architecture code* that implements a specific type of application or plugin format. The compiler can optionally generate *SIMD* code using either ALTIVEC or SSE2 intrinsics.

5.2 Implementations and performances

An implementation of the Faust compiler, written in C++, is available at *sourceforge* (http://faudiostream.sourceforge.net). The code is quite portable and only depends on Lex and Yacc for the parser code.

The performance of the code generate by the compiler is quite good. In order to compare it with hand written code we have reimplemented in Faust two audio effects:*Freeverb*, a well known reverb written in C++, and Tapiir a multitap delay also written in C++. Both applications are freely available on Internet with the source code.

In both case, the Faust specification is far more compact than the C++ code. The speed of both versions of the Freeverb are equivalents (but the speed of the Faust version using SIMD code generation is about twice faster). The Faust version of Tapiir is twice faster than the original version even in scalar mode.

6 Conclusion

We have presented both some syntactical and semantical aspects of Faust. The syntax of Faust is quite surprising at first but it turns out to be very convenient. Once you get used to it, it is both expressive and readable. The combination of functional programming and block-diagram composition is also really pleasant and natural to use.

Because of it simple and well defined semantic, the language can be easily compiled into efficient C++ code. Preliminary performance tests on the generated code are encouraging. In some cases the Faust code significantly outperforms hand-written code.

The compiler can be improved in several ways, in particular by extending its capacities of symbolic simplification and normalization and by improving the generation of SIMD code.

The semantic of the language can also be enlarged. Right now Faust only deals with scalar signals. The next step is to add arbitrary data types in particular vectors and matrices which are needed for image and video processing as well as spectral based transformations.

References

1. J. B. Dennis and D. P. Misunas. A computer architecture for highly parallel signal processing. In *Proceedings of the ACM 1974 National Conference*, pages 402–409. ACM, November 1974.
2. G. Kahn. The semantics of a simple language for parallel programming. In *Proceedings of the IFIP Congress 74*. North-Holland, 1974.
3. K. Karplus and A. Strong. Digital synthesis of plucked-string and drum timbres. *Computer Music Journal*, 7(2):43–55, 1983.
4. E. A. Lee and T. M. Parks. Dataflow process networks. In *Proceedings of the IEEE*, volume 83, pages 773–801, May 1995.
5. Y. Orlarey, D. Fober, and S. Letz. An algebra for block diagram languages. In ICMA, editor, *Proceedings of International Computer Music Conference*, pages 542–547, 2002.
6. Gheorghe Stefanescu. The algebra of flownomials part 1: Binary flownomials; basic theory. Report, Technical University Munich, November 1994.
7. Robert Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997.