Syntactical and Semantical Aspects of Faust

Yann Orlarey, Dominique Fober, Stephane Letz

April 27, 2004

Abstract

A draft and incomplete version of the paper...

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 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 factorisations to produce efficient code.

The first section describes the block-diagram algebra. Then we will present the Faust primitives and we will end with a concrete example.

2 The block-diagram algebra

Block-diagram formalisms are widely used in visual languages particularly musical languages. The user creates programs (i.e. block-diagrams), by connecting graphical *blocks*, representing the functionalities of the system. In almost every implementation, a block-diagram are represented internally as a graph, and interpreted as a *dataflow* computation (see [2] and[1] for historical papers on dataflow, and [5] or [3] for surveys).

This very common approach has several drawbacks. First, 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 actual behaviour of a block-diagram can be difficult to understand without a good knowledge of the implementation.

Secondly, dataflow models are difficult to implement efficiently. 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 plugins.

Thirdly, graphs are complex to manipulate. For example it might be desiderable 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 also 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* [4]. This algebra is independant from any particular language and of any particular meaning associated with the connections or the building blocks.

2.1 Definitions

Let $\mathbb{D}$ be the set of block-diagrams. A block-diagram $D \in \mathbb{D}$ is either an *identity* block (_), a *cut* block (!), a primitive block $b \in \mathbb{B}$, or any composition of two block-diagrams :

$$D = \_ \mid ! \mid B \mid (D_1 : D_2) \mid (D_1, D_2) \mid (D_1 <: D_2) \mid (D_1 :> D_2) \mid (D_1 \sim D_2) \tag{1}$$

A block-diagram D has I_D inputs and O_D outputs thus notated :

$$\text{inputs}(D) = \{D_{in}[0], D_{in}[1], \ldots, D_{in}[I_D - 1]\} \tag{2}$$

$$\text{outputs}(D) = \{D_{out}[0], D_{out}[1], \ldots, D_{out}[O_D - 1]\} \tag{3}$$

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.

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 ouputs) of (A, B) are the inputs (resp. the outputs) of A and B.

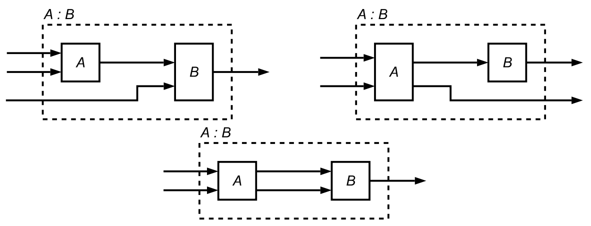

Figure 1: The sequential composition operator : possible cases according to the number of outputs of A and inputs of B

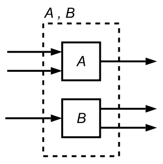

Figure 2: The parallel composition operator

2.4 Split composition $(A <: B)$

This *split composition* operator is used to distribute the outputs of A to several inputs of B. 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 $I_B = n.O_A$ then $A_{out}[i]$ is connected to $B_{in}[i + j.O_A]$ where $j < n$.

The inputs (resp. the outputs) of $(A <: B)$ are the inputs of A (resp. the outputs of B).

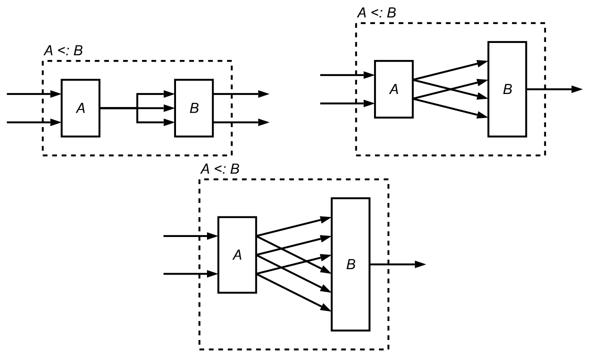

Figure 3: The split composition operator

Note that if $O_A = I_B$, 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. The general rule is that $A_{out}[i + j.I_B]$ is connected to $B_{in}[i]$ where $j < n$ and $O_A = n.I_B$. In Faust's signal processing semantic, merged connections correspond to added signals.

The inputs (resp. the outputs) of $(A :> B)$ are the inputs of A (resp. the outputs of B).

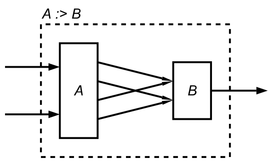

Figure 4: The merge composition operator

Note that if $O_A = I_B$, 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 (via an implicit 1 sample delay in Faust signal processing semantic, represented by a circle on the diagrams). 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.

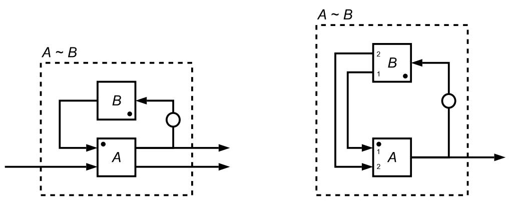

Figure 5: Two exemples of *recursive composition*

2.7 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 :

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

instead of

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

Moreover, the following properties hold :

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

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

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

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

3 Faust Primitive Boxes

The set of Faust primitives as been designed to follow C/C++ operations and to be as much as possible independant of any musical semantics. Typical musical operations, like oscillators, variable delays, filters, etc. are provided as external libraries written in Faust. A limited set of User Interface primitives, independent of any toolkit, is also provided.

Before going into the details of these primitives we need some few definitions.

3.1 Notations and definitions

3.1.1 Signals

A *signal* s is a discrete function of time $s : \mathbb{N} \rightarrow \mathbb{R}$. $s(t)$ represents the value of signal s at time t. The interval $[-1, +1]$ corresponds to the full range of the AD/DA converters. We define $\mathbb{S}$ to be the set of all possible signals : $\mathbb{S} = \mathbb{N} \rightarrow \mathbb{R}$.

A signal is *constant* if $\exists k, \forall t, s(t) = k$. We notate $\mathbb{S}_c \subset \mathbb{S}$ the subset of constant signals.

3.1.2 Signal processors

Faust primitives (and Faust block-diagrams in general) represent *signal processors*, functions transforming some *input signals* into *output signals*. A signal processor $p \in \mathbb{P}$ is a function from n-tuples of signals to m-tuples of signals $p : \mathbb{S}^n \rightarrow \mathbb{S}^m$. We will write $(x_1, \ldots, x_n)$: a n-tuple of signals of $\mathbb{S}^n$, and $()$: the empty tuple, single element of $\mathbb{S}^0$.

3.1.3 Semantic function

In order to refer explicitely to the mathematical *meaning* of a block-diagram D and to distinguish it from its syntactic representation we will used the notation $[\![D]\!]$ which means : *the signal processor represented by block-diagram D.*

3.2 Identity box _ and Cut box !

As shown in figure 6, the *identity* primitive (_) is essentially a simple wire representing the identity function for signals :

$$\begin{aligned} [\![\_]\!] &: \mathbb{S} \rightarrow \mathbb{S} && (4)\\ [\![\_]\!](s) &= (s) && (5) \end{aligned}$$

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

$$\begin{aligned} [\![\,!\,]\!] &: \mathbb{S} \rightarrow \mathbb{S}^0 && (6)\\ [\![\,!\,]\!](s) &= () && (7) \end{aligned}$$

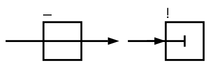

Figure 6: The *identity* and *cut* primitive boxes

3.3 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 :

$$\begin{aligned} [\![\star]\!] &: \mathbb{S}^2 \rightarrow \mathbb{S} && (8)\\ [\![\star]\!](s_1, s_2) &= (y) && (9)\\ y(t) &= s_1(t) \star s_2(t) && (10) \end{aligned}$$

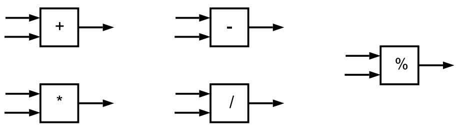

Figure 7: The arithmetics primitives

3.4 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 :

$$\begin{aligned} [\![\bowtie]\!] &: \mathbb{S}^2 \rightarrow \mathbb{S} && (11)\\ [\![\bowtie]\!](s_1, s_2) &= (y) && (12)\\ y(t) &= \begin{cases} 1 & \text{if } s_1(t) \bowtie s_2(t) \\ 0 & \text{else} \end{cases} && (13) \end{aligned}$$

3.5 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 :

$$\begin{aligned} [\![\star]\!] &: \mathbb{S}^2 \rightarrow \mathbb{S} && (14)\\ [\![\star]\!](s_1, s_2) &= (y) && (15)\\ y(t) &= s_1(t) \star s_2(t) && (16) \end{aligned}$$

3.6 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

$$\begin{aligned} [\![k]\!] &: \mathbb{S}^2 \rightarrow \mathbb{S} && (17)\\ [\![k]\!]() &= (y) && (18)\\ y(t) &= k && (19) \end{aligned}$$

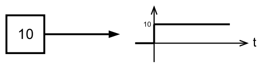

Figure 8: The constant 10

3.7 Casting

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

Figure 9: The *int cast* and *float cast* primitive boxes

3.8 Foreign definitions

Foreign definitions are used to incorporate externally defined C functions and constants. Foreign function are declared using the reserved word `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>)
```

3.9 Tables and delays

3.9.1 Fixed delays

Fixed delays are provided with two primitives `mem` and `@`. More sophisticated delays are implemented using the read-write tables. The `mem` represent a 1-sample delay :

$$\begin{aligned} [\![\text{mem}]\!] &: \mathbb{S} \rightarrow \mathbb{S} && (20)\\ [\![\text{mem}]\!](x) &= (y) && (21)\\ y(t+1) &= x(t) && (22) \end{aligned}$$

While `@` represent a fixed delay :

$$\begin{aligned} [\![@]\!] &: \mathbb{S}^2 \rightarrow \mathbb{S} && (23)\\ [\![@]\!](x,d) &= (y) && (24)\\ y(t+d) &= x(t) && (25) \end{aligned}$$

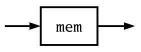

Figure 10: The *mem* boxe represents a one sample delay

3.9.2 Read-only table

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

$$\begin{aligned} [\![\text{rdtable}]\!] &: \mathbb{S}^3 \rightarrow \mathbb{S} && (26)\\ [\![\text{rdtable}]\!](n,v,i) &= (y) && (27)\\ y(t) &= v(i(t)) && (28) \end{aligned}$$

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

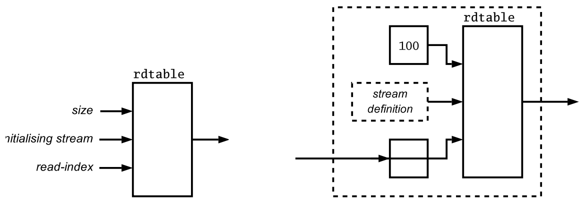

Figure 11: The read-only table primitive

3.9.3 Read-write table

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

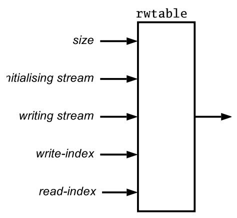

Figure 12: The read-write table primitive

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.

$$\llbracket \text{select2} \rrbracket : \mathbb{S}^3 \to \mathbb{S} \tag{29}$$
$$\llbracket \text{select2} \rrbracket(i, s[0], s[1]) = (y) \tag{30}$$
$$y(t) = s[i(t)](t) \tag{31}$$

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 :

$$\llbracket \text{select3} \rrbracket : \mathbb{S}^4 \to \mathbb{S} \tag{32}$$
$$\llbracket \text{select3} \rrbracket(i, s[0], s[1], s[2]) = (y) \tag{33}$$
$$y(t) = s[i(t)](t) \tag{34}$$

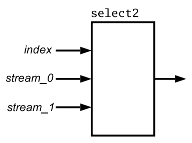

Figure 13: The *select2* primitive boxes

3.11 Graphic user interface

A Faust block-diagram can describe its own user interface using *buttons*, *sliders* and *groups* independently of any user interface toolkit. Buttons and sliders are essentally signals that vary according to the user actions. Groups are used to define the layout of the user interface.

3.11.1 Button

The `button` primitive has the following syntax :

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

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.

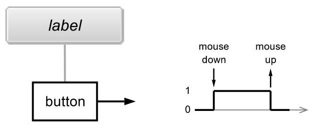

Figure 14: The *button* primitive boxes

3.11.2 Checkbox

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

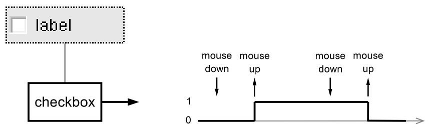

Figure 15: The *checkbox* primitive boxes

Here's the syntax :

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

3.11.3 Sliders

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 produce 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.

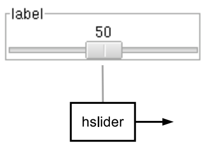

Figure 16: The *slider* primitive boxes

3.11.4 Numeric entry

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*)

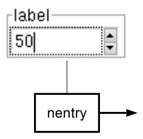

Figure 17: The *nentry* primitive boxes

3.11.5 Groups

The primitives `hgroup`, `vgroup` and `tgroup` are used to define the layout of the user interface.

hgroup("*label*", *Faust expression*)

`hgroup` defines an horizontal layout, while `vgroup` a vertical layout and `tgroup` a tabs organisation.

The following example results in the figure 18 when compiled for GTK+.

```
A = hslider("Slider 1",50,0,100,1);
B = hslider("Slider 2",50,0,100,1);
C = hslider("Slider 3",50,0,100,1);
process = tgroup("Sliders",(A,B,C));
```

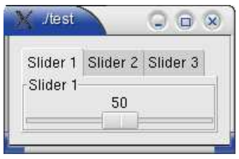

Figure 18: An example of tgroup layout

4 The Faust Compiler

5 An Example

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] E. A. Lee and T. M. Parks. Dataflow process networks. In *Proceedings of the IEEE*, volume 83, pages 773–801, May 1995.

[4] 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.

[5] Robert Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997.