# Compose with Faust in the Web

Sarah Denoux
GRAME
11 cours de verdun (gensoul)
69002 Lyon, France
sdenoux@grame.fr

Yann Orlarey
GRAME
orlarey@grame.fr

Stephane Letz
GRAME
letz@grame.fr

Dominique Fober
GRAME
fober@grame.fr

## ABSTRACT

The intention to provide recombinant components that can be selected and assembled in various and arbitrary combinations is shared by many. This idea of composability is present in several numerical domains, such as the Web or the audio community. The FAUST DSP audio language, designed with this concept, explores the idea of composing FAUST programs at a macro level, taking advantage of the Web as a deployment platform.

The Web offers a great opportunity to share, deploy and use programs without installation difficulties. So that having tools to deploy and compose FAUST DSPs online is a decisive challenge for it opens up a large panel of use cases. The network could now be used to reuse existing FAUST code, test it easily, combine it with someone else's code, redeploy the resulting DSP in a different environment, etc.

This article presents FAUST principles for composability and the implemented tools to compose FAUST programs within the Web.

## Keywords

FAUST, Composability, Web, DSP programming

## 1. INTRODUCTION

The composition of functions in mathematics has been defined for a long time as the operation that takes the output of one function as the input of a second one.

On the Web, the idea of composability is at the center of attention. The reflexion about how to build bridges between pages and compose data have brought links, iframe, rss streams, etc. Recently, Web Components have made their apparition, trying to provide a higher level of composability for the HTML language. A tool like Visual Composer for wordpress is an excellent example of HTML composition by simple drag and drop, creating complex pages from simple elements with a very transparent interface [**?**].

In the audio domain, a way to compose applications is through the concept of audio graphs that is used by many audio environments (Max/MSP, JACK, Supercollider, the Web audio API, etc), connecting the ouputs of a node to the inputs of another one.

The goal of this article is to present FAUST principles for composability and their uses to implement tools that can compose FAUST programs in the web. It will first describe the necessary principles of FAUST and its compiler (Section **??**) Then, two approches of FAUST composition will be explored (Section **??**), to finally present the provided interface for online composition of FAUST programs (Section **??**).

## 2. FAUST - A COMPOSABLE LANGUAGE

### 2.1 Faust

FAUST [Functional Audio Stream] [**?**] [**?**] is a functional, synchronous, domain-specific programming language specifically designed for real-time signal processing and synthesis. A unique feature of FAUST is that programs are fully compiled and can be deployed in many environments (Max/MSP, VST, ...) and platforms (OSX, Android, Linux, ...).

### 2.2 Design principles

FAUST is a specification language. It aims to provide an adequate notation to describe signal processors from a mathematical point of view and at a sample-precision level.

FAUST is, as much as possible, free from implementation details.

The semantics of FAUST allows the FAUST compiler to be semantically driven. Instead of compiling a program literally, it compiles the mathematical function it denotes, resulting in an optimized program.

Moreover, FAUST programs are signal processors that treat only one type of data, making it really easy to compose the output of a processor with the input of another one.

### 2.3 Semantics

FAUST is a textual language but nevertheless block-diagram oriented. It actually combines two approaches: functional programming and algebraic block- diagrams. The key idea is to view block-diagram construction as function composition. For that purpose, FAUST relies on a block-diagram


1Âř Web Audio Conference

algebra of five composition operations:

- Sequential composition, ":"
- Parallel composition, ","
- Recursive composition, " "
- Merge, ":>"
- Split, "<:"

One can think of each of these composition operations as a particular way to connect two block diagrams. Requirements about the number of channels of two composed block diagrams are bounding the specification. For example for the sequential composition, A : B, the following rule is specified:

$$numOutPuts(A) = numInputs(B).$$

# 3. TWO APPROACHES TO THE COMPOSITION OF FAUST PROGRAMS

## 3.1 Graph composition

The notion of audio graph is, in the numerical domain, what replaced the analogic way of patching audio modules with cables. This concept is widely shared between the audio community (JACK, Max, Supercollider, the Web Audio API, etc), connecting the ouputs of a node to the inputs of another one. The communication between nodes of such a graph is possible at the condition that the data types flowing are compatible.

Faust programs don't go against it and can be part of a graph through those environments. It can already be tested by deploying a FAUST program within JACK, Max/MSP or Supercollider. In the Web, this idea could be implemented within a page where FAUST programs would be webaudio API nodes patched together to create a graph.

This model is one way to compose Faust programs. But this approach has two downsides :

- External tools are needed to create the connection between programs
- The resulting patch is not optimized as it would be if compiled through Faust compiler

This solution will be hopefully implemented before the camera-ready version of the paper, enabling a comparison between the two approaches with benchmarks.

## 3.2 Equivalent Faust Composition

Another approach for composing FAUST programs is to calculate an equivalent FAUST program from a graph of applications.

Within a FAUST program, two FAUST expressions can be very easily composed with the operators described in **??**. But a FAUST program is a list of statements, definitions, etc. So that compose two programs has more to it. Therefore the keyword "component" has made its apparition into FAUST syntax. It allows to reuse a full FAUST program as a simple expression.

*component("karplus32.dsp")*

It can then be composed with other components using the basic operators :
*process = component("karplus32.dsp") : component("echo.dsp");*

Creating a tool to implement that type of composition is the guideline of the next section.

# 4. COMPONENT CREATOR

Following the described use case (cf. **??**), the idea of the component creator is to have a tool to compose FAUST programs at a "Macro" level. Once a visual patch is assembled by simple drag and drop, the user can compile the resulting component : the full equivalent DSP expression is created, following the semantics and layout requirements (Section **??** and **??**), compiled and can run fully optimized thanks to Faust compiler.

## 4.1 Use Case

A FAUST user prototypes a physical model of guitar and wants to share it. So, using faust2webaudioasm (cf. **??**) (manually or through FaustLive), he deploys his DSP as an HTML page. By adding this page to his server, anybody can play his guitar.

Another user, surfing on the Web, comes on the FAUST guitar and wants to test it by combining it with the distortion he just implemented. A simple test would be to put them in sequence. So, he opens the component-creator page on FAUST website or in his FAUSTLive, drops the guitar then his distortion and creates a component. The DSPs are combined, compiled and the resulting DSP now runs. And he can then deploy it as a new native HTML page or as any other kind of application/plugin supported by the FAUST project.

## 4.2 Faust tools for the Web

The internet provides a world wide deployment platform for applications and services. Taking advantage of this opportunity, the FAUST project is deploying on the Web. Not only can users deploy their DSPs on Web pages, but they now have the tools to dynamically test and combine their programs online.

### 4.2.1 faust2webaudioasm

Thanks to its JavaScript backend, the FAUST Compiler is now able to compile a FAUST program into JavaScript. A new tool is therefore available in the FAUST distribution : faust2webaudioasm. Thanks to this tool, FAUST developers can now create their DSPs and share them by deploying them as Web pages. They can whether do it manually with the FAUST command line or use FaustLive [**?**] to export their program as an HTML page.

### 4.2.2 libfaust.js

Emscripten is a technology of compiler that facilitates the port of large C/C++ code base in JavaScript. The FAUST compiler, written in C++, benefitted from that technology to create an asmjs version of the compiler, called libfaust.js. It is therefore possible to embedded the FAUST compiler into a Web page and compile arbitrary FAUST code on the client-side.

Published on faust server [?], this library can be easily imported in a page, permitting, for example, online composition of FAUST programs.

## 4.3 The Component Creator Interface

The tool available in FaustLive, provides a simple interface:

- *columns* represent parallel composition
- *rows* represent sequential composition
- a zone for a feedback element is available.

In each zone, you can drop different types of elements: dsp files, strings, Web urls, ... Each of them represents a FAUST DSP. Columns and rows can be added/deleted at will (Figure ??). Once a visual combination is fullfilled, the "create component" button starts the process that creates the equivalent FAUST program, corresponding to the visual composition.

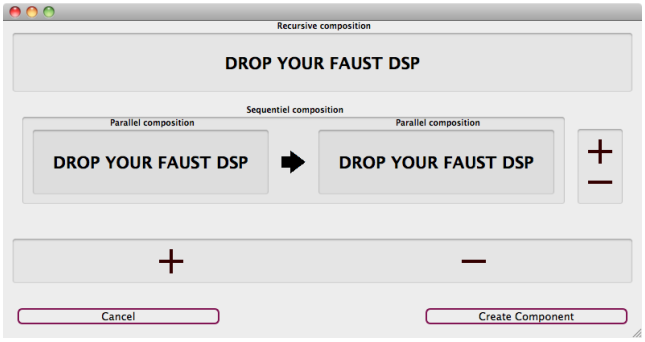


**Figure 1: FaustLive component creator**

The Web implementation has the same semantics and layout as the one in FaustLive. But as any Web application, it has the advantage to be accessible by anybody without any primary installation. This page, implemented in HTML and JavaScript is based on libfaust.js (cf. ??). The use case (cf. ??) executed on the Web interface is described on Figure ?? and ??.

## 4.4 Semantics

A brake to FAUST composability lies in its semantics: in order to combine two DSPs, their number of channels have to match. Considering the desired use case, the idea is to be able to compose with someone else's DSPs very easily. For that matter, having to know the channel characteristics of a DSP is not desirable. Therefore, it was chosen to create stereo DSPs independently from their original characteristics. Two new functions were introduced as part of music.lib: stereoize and recursivize. Stereoize provides the required processing to transform an arbitrary DSP into a stereo effect with two inputs and two outputs (Figure ??). Recursivize does the same for two arbitrary DSP to be composed with recursivity.

Moreover, when there are both parallel and sequential compositions to be executed, the choice was to: first perform the parallel composition of each item of a column then stereoize the result and finally add the sequential composition. In case a recursive element is added, the recursive com-

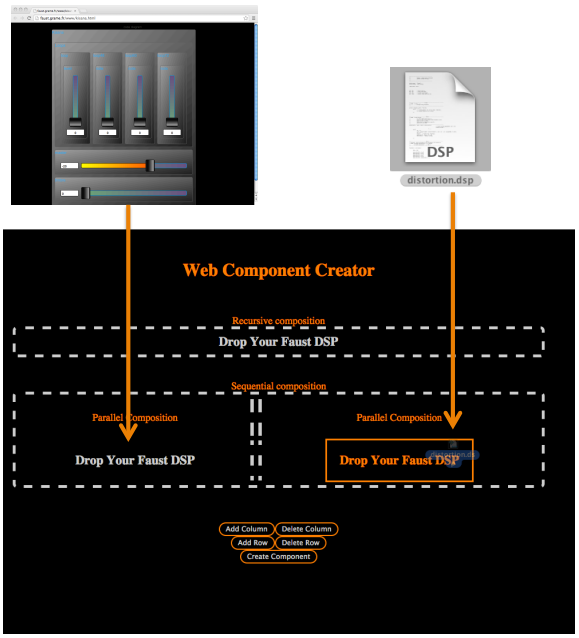


**Figure 2: Drop on HTML component creator**

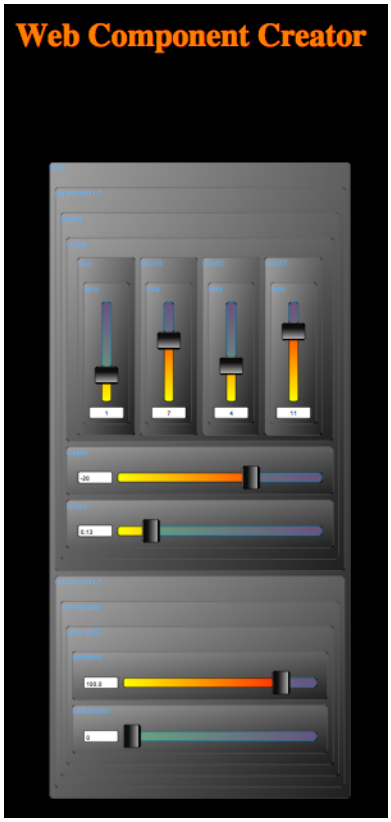


**Figure 3: HTML resulting component**

$$
\begin{array}{c|l}
 & \textit{degenerated processor} \\
 & S(n,0) =!,! : 0,0; \\
 & \\
 & \textit{processors with no inputs} \\
 & S(0,1) =!,! : p <: \_,\_; \\
 & S(0,2) =!,! : p; \\
 & S(0,n) =!,! : p, p :> \_,\_; \\
 & \\
stereoize(p) = & \textit{processors with one input} \\
S(inputs(p), outputs(p)) & S(1,1) = p,p; \\
 & S(1,n) = p, p :> \_,\_; \\
 & \\
 & \textit{processors with two inputs} \\
 & S(2,1) = p <: \_,\_; \\
 & S(2,2) = p; \\
 & \\
 & \textit{othercases} \\
 & S(n,m) = \_,\_ <: p, p :> \_,\_;
\end{array}
$$

**Figure 4: Faust formula of the Stereoize function**

position comes at last. For example, if we have the visual composition shown on Figure **??**, the resulting component will be:

*process = recursivize((stereoize(A,C) : stereoize(B,D)),E)*

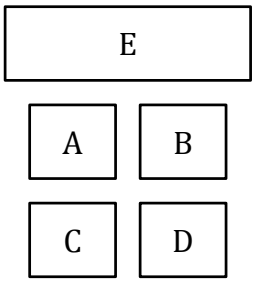


**Figure 5: Example of visual configuration**

## 4.5 Layout Composition

A FAUST program denotes a processing algorithm but also provides an abstract layout specification. For that matter, when composing FAUST programs, layouts also have to be composed.

### 4.5.1 Specification

The model of FAUST layout is simple. A hierarchy of groups (vertical or horizontal) can be described in the code. Then, inside a same level of hierarchy, elements are sorted alphabetically.

The problem brought up by that model, is that when you compose two FAUST programs, interface elements like sliders, buttons, etc can be mixed and result in an interface that doesn't reflect the original composition.

The outcoming specification for FAUST component creator is to respect the algorithmic order of the items:

If the component is *(A,B):(C,D),* D cannot be layed out next to the item A.

The other chosen constraint is to optimize the occupied zone on the screen, by organizing the layout with horizontal/vertical groups.

### 4.5.2 Implementation

The implementation is based on a binary tree were each node has a notion of its layout:

- leaf nodes: containing the real FAUST interface (included in a group, called "component-i.j")
- vertical containers, putting in a vertical group its two sub-nodes

$$\frac{T_1}{T_2}$$

- horizontal containers, putting in a horizontal group its two sub-nodes

$$T_1|T_2$$

The construction of the tree is trying to reduce the surface of the interface. The best disposition is calculated in the list of possible trees.

- The function B calculates the final and best disposition from the list of possible trees

$$B(C(< T_{\text{i}} >)) = \begin{cases} T_0 & S(T_0) < S(B(< T_{\text{i-1}} >)) \\ B(< T_{\text{i-1}} >) & otherwise \end{cases}$$

- The function C creates the list of possible trees from the list of components, where components are the leaf nodes

$$C(< T_{\text{i}} >) = \begin{cases} empty & if \;\; size(< T_{\text{i}} >) = 0 \\ T_0 & if \;\; size(< T_{\text{i}} >) = 1 \\ M(T_0, T_1) & if \;\; size(< T_{\text{i}} >) = 2 \\ D(T_0, < T_{\text{i-1}} >) & otherwise \end{cases}$$

- The function D dispatches a component on a list of trees, creating two new trees from one. An exemple is given on figure **??**.

$$D(I, < T_{\text{i}} >) =< T_{1\text{i}}, T_{2\text{i}} >$$

$$With \;\; T_{1\text{i}} = M(M(I, T_{\text{ileft}}), T_{\text{iright}}) T_{2\text{i}} = M(I, T_{\text{i}})$$

- The function M chooses the type of container (vertical or horizontal) wrapping two trees: T1 and T2

$$M(T_1, T_2) = \begin{cases} T_1|T_2 & if \;\; surface(T_1|T_2) < surface(\frac{T_1}{T_2}) \\ \frac{T_1}{T_2} & otherwise \end{cases}$$

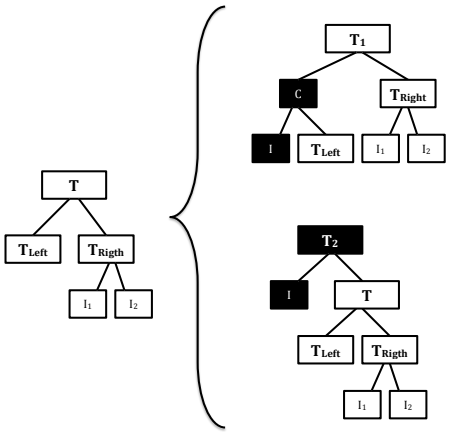


**Figure 6: Dispatch an element, I, on a tree, T**

# 5. KNOW ISSUES

Some improvements still have to be made :

- The FAUST equivalent could be recompiled automatically with every change, for the user to hear the result of his composition dynamically.
- A DSP with dependencies cannot yet be compiled in the Web component creator, because the libraries cannot be found. FAUST libraries should be available on the Web and be imported that way in FAUST programs.
- As seen in section **??**, the merge (:>) and split (<:) operators are not part of component creator semantics. Along with a more complex interface, it could be possible to integrate them in the future.
- In a really near future, implementing the webaudio composition and compare with benchmarks the performances of the two approaches.

# 6. CONCLUSION

Deploy, Share, Compose FAUST programs in the Web is now at hand, amplifying he panel of possibility for FAUST usage.

**Acknowledgments**

This work has been implemented under the FEEVER project [ANR-13-BS02-0008] supported by the “Agence Nationale pour la Recherche”.