A Rationale for Faust Design Decisions

Y. Orlarey[1], D. Fober[1], S. Letz[1], A. Gräf[2], and P. Jouvelot[3]

[1]Grame, centre national de création musicale, France
[2]Computer Music Research Group, University of Mainz, Germany
[3]MINES ParisTech, PSL Research University, France

Musicians have been using specifically designed programming languages since Music III (Mathews, 1959) and MUSICOMP (Hiller and Baker, 1963). Today, music DSLs like Csound, OpenMusic, Max, Puredata, Faust or Supercollider, to name a few, are routinely used as a creative means by electronic musicians, sound engineers and avant-garde composers.

In this presentation we outline various decisions we took in the design of Faust [Functional Audio Stream][1,2], a DSL used on stage for concerts and artistic productions, in education and research, as well as in open sources projects and commercial applications.

Focused on signal processing

Some music DSLs are DSP-oriented (digital signal processing), some others are music composition-oriented, and some try to combine both approaches. Faust is in the first category, with a strong focus on the design of synthesizers, music instruments, audio effects, etc.. This focused approach allows Faust to be based on a simple model, the notion of *signal processors*. Everything in Faust is a signal processor and programming in Faust consists in composing signal processors together.

A language for end users

Users of music DSLs are typically musicians, sound engineers, researchers, musical assistants, etc. They often have a background in signal processing or at least a clear idea of how audio effects and sound synthesis systems should work or sound. But they are not necessarily computer scientists or professional developers. The development of real-time audio software in C is usually out of reach for most of them. The ambition of Faust is to offer them a viable and efficient high-level alternative to C/C++ to describe and implement high-performance musical instruments, audio effects and more generally real-time signal processing applications.

Compiled instead of interpreted

Most music DSLs are interpreted languages. To amortize the interpretation cost these languages work on block of audio samples instead of individual samples. Therefore operations requiring to work at sample level, for instance IIR filters, can't be efficiently implemented in interpreted languages and have to be provided as primitives or as external plug-ins. The advantage of a fully compiled language like Faust is that it can be used to implement efficient sample-level DSP algorithms. Moreover the Faust compiler provides automatic parallelization as well as advanced optimization techniques that allow the generated code to compete with hand-written C code.

[1] `http://faust.grame.fr`
[2] `http://www.grame.fr/ressources/publications/faust-soft-computing.pdf`

A high-level specification language

Faust is designed to be a high-level *specification language* rather than an implementation language. We chose to make a clear separation between the users' role, in charge of the specification, and the role of the compiler, in charge of the implementation. The way the user writes a Faust program should not matter; only its meaning should count. Two different Faust programs, but with the same mathematical meaning, should result in the same implementation[3].

Simple well-defined formal semantics

Faust has a simple and well-defined formal semantics[4]. A Faust program denotes a signal processor: a continuous function that maps a tuple of input signals to a tuple of output signals. Having a simple and well defined semantics is not only of academic interest; it makes the language easier to learn and to use, and it allows the Faust compiler to do very advanced optimizations. Moreover it allows the generation of some automatic documentation and preservation techniques that are very useful in the field of computer music.

Purely functional approach

Functional programming provides Faust with a high level of modularity, both to compose and understand Faust programs. Moreover it offers a very natural framework for signal processing. Periodically-sampled digital signals can be modeled as functions of time. Signal processors are second-order functions operating on signals. Faust block-diagram algebra is a set of third-order composition operations on signal processors. Finally, user-defined functions are higher-order functions on block-diagram expressions.

Textual block-diagram-oriented syntax

Music DSLs can be roughly divided into textual (Csound, Supercollider, Faust) and visual (Max, Puredata) languages. But in both cases the concept of block-diagrams, functional blocks connected by signals, is usually central. Faust is a textual language with a concise block-diagram-oriented syntax. The syntax is based on an algebra of five signal processor composition operations. It is designed to favor modularity and composability of programs. It can be easily translated into visual block-diagrams.

Easy deployment

Musicians have to deal with a large variety of operating systems, software environments and hardware architectures. Faust is designed to favor an easy deployment of Faust programs on all these targets by making a clear separation between the computation itself, as described by the program text, and how this computation should be related to the external world. This relation (with audio drivers, GUI, sensors, etc.) is described in specific *architecture files*. These architecture files are essentially wrappers for the code generated by the compiler. Currently more than 40 architecture files are provided for the major computer operating systems as well as iOS, Android and some embedded systems (see `https://www.youtube.com/watch?v=ITOm-LzbjZI`).

Acknowledgments

This research was funded in part by the Agence nationale pour la recherche (ANR): Projects ASTREE [2008-CORD-003], INEDIT [ANR 2012 CORD 009] and FEEVER [ANR-13-BS02-0008].

[3] Although this is undecidable in general.

[4] `http://www.grame.fr/ressources/publications/faust-elsevier2011.pdf`