Numéros / n° 4 - automne 2014

« Faust Programs in Csound »

Victor Lazzarini

1. Introduction

Csound (Boulanger, 2000) is a Music-N language that was first released in 1986 as a C-port of the Music 11 language, originally developed for the DEC PDP-11 computers at MIT in the 1970s. Since its release, it has undergone a number of changes, and in 2006, a completely re-engineered system, Csound 5, was launched. This system had a number of new possibilities, as it was designed as a library that could be embedded in a variety of environments. It also included the possibility of plugin opcodes, which would extend the language without the need for the whole re-compilation of the code base. In 2013, a further review of the system was carried out, and a new major version, Csound 6 (Cabrera *et al.*, 2013), was launched, with substantial improvements and additions. This is the current system, discussed in this article.

Faust (Orlarey, 2009) is a functional language designed to translate signal processing flowcharts into C++ or Javascript source code, or into LLVM bitcode. It allows plugins to be designed and then translated to C++ code that can be compiled into dynamic libraries, which can then be loaded into Csound as new unit generators (opcodes). This is done by the Faust compiler, which can be executed from the command-line, from the IDE Faustworks or on-line, via a web-based frontend.

Recently, however, a new version of the system, Faust 2, has been developed where the Faust compiler is now provided as a library (libfaust) that can be embedded into another program. Libfaust can provide the Faust functionality in a dynamic way, where Faust programs can be compiled on-the-fly into LLVM bitcode that can be executed directly. This allows the possibility of short- circuiting the development process, so that a plugin opcode is not required anymore as an in-between the original Faust program and the running Unit generator in Csound.

2. Using the Faust library

There are three basic steps in the Faust dynamic compilation and performance:

1. Compilation of Faust code into a DSP Factory.
2. Instantiation of a DSP object from a DSP Factory.
3. Performance of the DSP process.

The first step employs one of the two functions

```
llvm_dsp_factory*

createDSPFactoryFromFile(const std::string& filename, int argc,
```

const char *argv[],

const std::string& target,

std::string& error_msg, int opt_level = 3);

.

llvm_dsp_factory*

createDSPFactoryFromString(const std::string& name_app, const

std::string& dsp_content,

int argc, const char *argv[], const

std::string& target,

std::string& error_msg, int opt_level = 3);

They translate the Faust code into a bitcode form that resides in memory and is ready to be executed in a program. In order to do so, we need to instantiate a DSP object from the factory:

```
llvm_dsp* createDSPInstance(llvm_dsp_factory* factory);
```

The DSP class has a number of public methods that can be used to manipulate it:

```
class llvm_dsp :public dsp {

    public:
     virtual int getNumInputs();
     virtual int getNumOutputs();
     virtual void init(int samplingFreq);
     virtual void buildUserInterface(UI* inter);
     virtual void compute(int count,
          FAUSTFLOAT** input, FAUSTFLOAT** output);
};
```

Finally, with a DSP object created, we can execute it by invoking its compute() method. This will consume a block of input samples and produce a block of output samples. To control the synthesis/processing, we can also set controls via a user interface object of our own design. This can be added to the functionality of the DSP object via the `buildUserInterface()` method. In the case of Csound, this is used to pass control data from Csound to the Faust program.

3. The Opcodes

The Csound Faust opcodes (see *The Csound Reference Manual*) are built using the above functionality. In order to facilitate their use, they split the steps into two: compilation and performance.

In Csound, initialisation and DSP performance are separated into two distinct phases. During the former, all unit generators that have initialisation tasks to execute are run (once), and this is followed by the latter

phase, which is a loop that invokes the performing functions of each signal-processing opcodes. Some opcodes do not have these functions, as they are not signal generators or consumers (or both). These opcodes run only at initialisation time.

The compilation step in Faust is an eminently initialisation-time affair. It does not involve any performance tasks (by which I mean signal processing). It makes sense to make it an init-time only opcode. Separating it from a DSP performance opcode also allows any number of DSP objects to be created from the same factory.

The performance aspect of the Faust integration is subdivided into two elements: signal processing and controls, and exists in two separate opcodes. The signal processing part is only concerned in picking up any input audio (if it exists), doing something it and passing it to the output. In order to do parameter control, we use another opcode that can adjust a given user interface element defined in the DSP.

Finally, there is also an opcode that has been designed for single-instance DSPs, which integrates a compilation phase at init-time and signal processing at perf-time. This opcode can be employed for "one-off" effects that are not designed to be run in multiple instances.

3.1. Faustcompile

The `faustcompile` opcode invokes the just-in-time compiler to produce an instantiable DSP process from a Faust program. It will compile a Faust program from a string, controlled by various arguments. Multi-line strings are accepted, using {{ }} to enclose the string.

3.1.1. Syntax

ihandle faustcompile Scode, Sargs

`Scode`? a string (in double-quotes or enclosed by {{ }}) containing a Faust program.

`Sargs`? a string (in double-quotes or enclosed by {{ }}) containing Faust compiler args.

Example:

ihandle faustcompile "process=+;", "-vec -lv 1"}

3.2. Faustaudio

The `faustaudio` opcode instantiates and runs a compiled Faust program. It will work with a program compiled with faustcompile

ihandle,a1[,a2,...] faustaudio ifac[,ain1,...]

`ifac`? a handle to a compiled Faust program, produced by faustcompile

`ihandle`? a handle to the Faust DSP instance, which can be used to access its controls with faustctl

`ain1,...`? input signals

`a1,...`? output signals

Example:

ifac faustcompile "process=+;", "-vec -lv 1"

idsp,a1 faustaudio ifac,ain1,ain2

3.3. Faustctl

The `faustctl` opcode adjusts UI controls in a Faust DSP instance. It will set a given control in a running Faust program.

3.3.1. Syntax

faustctl idsp,`Scontrol,kval`

`Scontrol`? a string containing the control name

`dsp`? a handle to an existing Faust DSP instance

`kval`? value to which the control will be set.

Example:

idsp,a1 faustgen {{

gain = hslider("vol",1,0,1,0.01);

process=(_ * gain);

}}, ain1

faustctl idsp, "vol", 0.5

3.4. Faustgen

The opcode `faustgen` compiles, instantiates and runs a compiled Faust program. It will invoke the just-in-time compiler at i-time, and instantiate the DSP program. At perf-time, it will run the compiled Faust code.

3.4.1. Syntax

`ihandle,a1[,a2,...]`faustgen SCode`[,ain1,...]`

`Scode`? a string containing a Faust program

`ihandle`? a handle to the Faust DSP instance, which can be used to access its controls with faustctl

`ain1,...`?input signals

`a1,...`?outputsignals

Example:

idsp,a1 faustgen "process=+;",ain1,ain2

4. Applications

A number of applications can be envisaged for Csound (and its frontends) with the new Faust opcodes:

- Synthesis: Csound can work as standalone software synthesiser, controlled by MIDI messages or by OSC commands. It can be used by MIDI sequencing software as the system synthesiser, and Faust programs can augment that. The classic CLI frontend, or any other frontend can be used for this.
- Plugin instruments: The Cabbage frontend can be used to generate VSTi plu- gins from Csound code. These can be loaded into any VST host program (such as sequencers, notation programs etc). Plugins can be created from standard Csound code, with no need of any extensions.
- Audio processor: Csound can be used as a realtime or offline audio processor.
- For realtime operation, Cabbage and CsLadspa can be used to generate plugins that can be loaded into VST or LADSPA hosts. The power of Faust can be used to augment Csound for the efficient implementation of custom processing algorithms.
- Custom software: Csound can also be used to create custom software applications, both for desktop operating systems and mobile (android, iOS). It can be programmed from a variety of levels, from the basic, top-level through Cabbage and csoundo, to middle-level, through Python, Clojure, Java, to lower-level through C and C++.

5. Conclusions

In this paper, we introduced the possibility of embedding Faust programs directly into the Csound orchestra. The details of the technology and its mechanisms have been explored, and we have shown how the Faust library is used in a series of opcodes supplied in a plugin library to Csound. Each opcode has been discussed separately and their syntax was presented. We conclude the exposition with a number of envisaged applications for the technology.

Pour citer ce document:
Victor Lazzarini, « Faust Programs in Csound », *RFIM* [En ligne], Numéros, n° 4 - automne 2014, Mis à jour le 14/10/2014
URL: http://revues.mshparisnord.org/rfim/index.php?id=361