DSP Programming with Faust, Q and SuperCollider

Yann ORLAREY
Grame, Centre National de Creation Musicale
Lyon, France
orlarey@grame.fr

Albert GRÄF
Dept. of Music Informatics
Johannes Gutenberg University
Mainz, Germany
Dr.Graef@t-online.de

Stefan KERSTEN
Dept. of Communication Science
Technical University
Berlin, Germany
stefan.kersten@tu-berlin.de

Abstract

Faust is a functional programming language for real-time signal processing and synthesis that targets high-performance signal processing applications and audio plugins. The paper gives a brief introduction to Faust and discusses its interfaces to Q, a general-purpose functional programming language, and SuperCollider, an object-oriented sound synthesis language and engine.

Keywords

Computer music, digital signal processing, Faust programming language, functional programming, Q programming language, SuperCollider

1 Introduction

Faust is a programming language for real-time signal processing and synthesis that targets high-performance signal processing applications and audio plugins. This paper gives a brief introduction to Faust, emphasizing practical examples rather than theoretic concepts which can be found elsewhere (Orlarey et al., 2004).

A Faust program describes a *signal processor*, a DSP algorithm that transforms input signals into output signals. Faust is a *functional* programming language which models signals as functions (of time) and DSP algorithms as higher-order functions operating on signals. Faust programs are compiled to efficient C++ code which can be included in C/C++ applications, and which can also be executed either as standalone programs or as plugins in other environments. In particular, in this paper we describe Faust's interfaces to Q, an interpreted, general-purpose functional programming language based on term rewriting (Gräf, 2005), and SuperCollider (McCartney, 2002), the well-known object-oriented sound synthesis language and engine.

2 Faust

The programming model of Faust combines a functional programming approach with a block-diagram syntax. The functional programming approach provides a natural framework for signal processing. Digital signals are modeled as discrete functions of time, and signal processors as second order functions that operate on them. Moreover Faust block-diagram *composition operators*, used to combine signal processors together, fit in the same picture as third order functions.

Faust is a compiled language. The compiler translates Faust programs into equivalent C++ programs. It uses several optimization techniques in order to generate the most efficient code. The resulting code can usually compete with, and sometimes outperform, DSP code directly written in C. It is also self-contained and doesn't depend on any DSP runtime library.

Thanks to specific *architecture files*, a single Faust program can be used to produce code for a variety of platforms and plugin formats. These architecture files act as wrappers and describe the interactions with the host audio and GUI system. Currently more than 8 architectures are supported (see Table 1) and new ones can be easily added.

alsa-gtk.cpp	ALSA application
jack-gtk.cpp	JACK application
sndfile.cpp	command line application
ladspa.cpp	LADSPA plugin
max-msp.cpp	Max MSP plugin
supercollider.cpp	Supercollider plugin
vst.cpp	VST plugin
q.cpp	Q language plugin

Table 1: The main architecture files available for Faust

In the following subsections we give a short and informal introduction to the language through two simple examples. Interested readers can refer to (Orlarey et al., 2004) for a more complete description.

2.1 A simple noise generator

A Faust program describes a signal processor by combining primitive operations on signals (like $+, -, *, /, \sqrt{}, \sin, \cos, \ldots$) using an algebra of high level *composition operators* (see Table 2). You can think of these composition operators as a generalization of mathematical function composition $f \circ g$.

f ~ g	recursive composition
f , g	parallel composition
f : g	sequential composition
f <: g	split composition
f :> g	merge composition

Table 2: The five high level block-diagram *composition operators* used in Faust

A Faust program is organized as a set of *definitions* with at least one for the keyword `process` (the equivalent of `main` in C).

Our noise generator example `noise.dsp` only involves three very simple definitions. But it also shows some specific aspects of the language:

```
random  = +(12345) ~ *(1103515245);
noise   = random/2147483647.0;
process = noise * checkbox("generate");
```

The first definition describes a (pseudo) random number generator. Each new random number is computed by multiplying the previous one by 1103515245 and adding to the result 12345.

The expression `+(12345)` denotes the operation of adding 12345 to a signal. It is an example of a common technique in functional programming called *partial application*: the binary operation `+` is here provided with only one of its arguments. In the same way `*(1103515245)` denotes the multiplication of a signal by 1103515245.

The two resulting operations are *recursively composed* using the ~ operator. This operator connects in a feedback loop the output of `+(12345)` to the input of `*(1103515245)` (with an implicit 1-sample delay) and the output of `*(1103515245)` to the input of `+(12345)`.

The second definition transforms the random signal into a noise signal by scaling it between -1.0 and +1.0.

Finally, the definition of process adds a simple user interface to control the production of the sound. The noise signal is multiplied by a GUI checkbox signal of value 1.0 when it is checked and 0.0 otherwise.

2.2 Invoking the compiler

The role of the compiler is to translate Faust programs into equivalent C++ programs. The key idea to generate efficient code is not to compile the block diagram itself, but *what it computes*.

Driven by the semantic rules of the language 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 normalized, and common subexpressions are factorized. Finally these expressions are translated into a self contained C++ class that implements all the required computation.

To compile our noise generator example we use the following command :

```
$ faust noise.dsp
```

This command generates the C++ code in Figure 1. The generated class contains five methods. `getNumInputs()` and `getNumOutputs()` return the number of input and output signals required by our signal processor. `init()` initializes the internal state of the signal processor. `buildUserInterface()` can be seen as a list of high level commands, independent of any toolkit, to build the user interface. The method `compute()` does the actual signal processing. It takes 3 arguments: the number of frames to compute, the addresses of the input buffers and the addresses of the output buffers, and computes the output samples according to the input samples.

The `faust` command accepts several options to control the generated code. Two of them are widely used. The option `-o` *outputfile* specifies the output file to be used instead of the standard output. The option `-a` *architecturefile* defines the architecture file used to wrap the generate C++ class.

For example the command `faust -a q.cpp -o noise.cpp noise.dsp` generates an external object for the Q language, while `faust -a jack-gtk.cpp -o noise.cpp noise.dsp` generates a standalone Jack application using the GTK toolkit.

Another interesting option is `-svg` that generates one or more SVG graphic files that represent the block-diagram of the program as in Figure 2.

```
class mydsp : public dsp
{
 private:

  int     R0_0;
  float   fcheckbox0;

 public:

  virtual int getNumInputs()  {
    return 0;
  }
  virtual int getNumOutputs() {
    return 1;
  }
  virtual void init(int samplingFreq) {
    fSamplingFreq = samplingFreq;
    R0_0 = 0;
    fcheckbox0 = 0.0;
  }
  virtual void buildUserInterface(UI* ui) {
    ui->openVerticalBox("faust");
    ui->addCheckButton("generate",
                              &fcheckbox0);
    ui->closeBox();
  }
  virtual void compute (int count,
           float** input, float** output) {
    float* output0; output0 = output[0];
    float ftemp0 = 4.656613e-10f*fcheckbox0;
    for (int i=0; i<count; i++) {
      R0_0 = (12345 + (1103515245 * R0_0));
      output0[i] = (ftemp0 * R0_0);
    }
  }
};
```

Figure 1: The C++ implementation code of the noise generator produced by the Faust compiler

2.3 The Karplus-Strong algorithm

Karplus-Strong is a well known algorithm first presented by Karplus and Strong in 1983 (Karplus and Strong, 1983). Whereas not completely trivial, the principle of the algorithm is simple enough to be described in few lines of Faust, while producing interesting metallic plucked-string and drum sounds.

The sound is produced by an impulse of noise that goes into a resonator based on a delay line with a filtered feedback. The user interface contains a button to trigger the sound production, as well as two sliders to control the size of both the resonator and the noise impulse, and the amount of feedback.

2.3.1 The noise generator

We simply reuse here the noise generator of the previous example (subsection 2.1).

```
random  = +(12345) ~ *(1103515245);
```

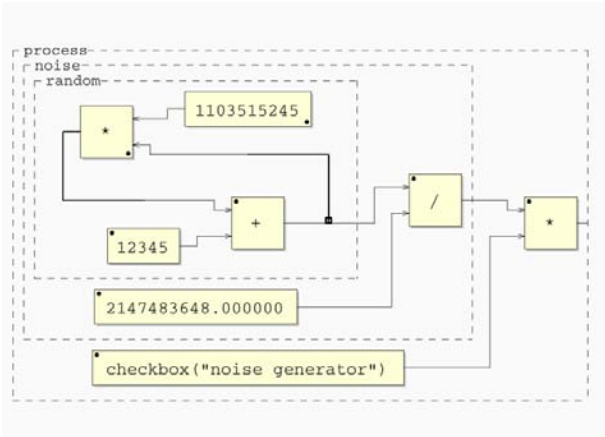

Figure 2: Graphic block-diagram of the noise generator produced with the -svg option

```
noise   = random/2147483647.0;
```

2.3.2 The trigger

The `trigger` is used to transform the signal delivered by a user interface button into a precisely calibrated control signal. We want this control signal to be 1.0 for a duration of exactly n samples, independentely of how long the button is pressed.

```
impulse(x) = x - mem(x) : >(0.0);
decay(n,x) = x - (x>0.0)/n;
release(n) = + ~ decay(n);
trigger(n) = button("play") : impulse
           : release(n) : >(0.0);
```

For that purpose we first transforms the button signal into a 1-sample impulse corresponding to the raising front of the button signal. Then we add to this impulse a kind of *release* that will decrease from 1.0 to 0.0 in exactly n samples. Finally we produce a control signal which is 1.0 when the signal with release is greater than 0.0.

All these steps are combined in a four stages sequential composition with the operator ':'.

2.3.3 The resonator

The resonator uses a variable delay line implemented using a table of samples. Two consecutive samples of the delay line are averaged, attenuated and fed back into the table.

```
index(n)    =  &(n-1) ~ +(1);
delay(n,d,x)= rwtable( n, 0.0, index(n),
            x, (index(n)-int(d))&(n-1) );
average(x)  = (x+mem(x))/2;
resonator(d,a) = (+ : delay(4096, d-1))
               ~ (average : *(1.0-a));
```

2.3.4 Putting it all together

The last step is to put all the pieces together in a sequential composition. The parameters of the trigger and the resonator are controlled by two user interface sliders.

```
dur = hslider("duration",128,2,512,1);
att = hslider("attenuation",
                          0.1,0,1,0.01);
process = noise
        : *(trigger(dur))
        : resonator(dur,att);
```

A screen shot of the resulting application (compiled with the `jack-gtk.cpp` architecture) is reproduced in Figure 3. It is interesting to note that despite the fact that the duration slider is used twice, it only appears once in the user interface. The reason is that Faust enforces *referential transparency* for all expressions, in particular user interface elements. Things are uniquely and unequivocally identified by their definition and naming is just a convenient shortcut. For example in the following program, `process` always generate a null signal:

```
foo = hslider("duration", 128, 2, 512, 1);
faa = hslider("duration", 128, 2, 512, 1);
process = foo - faa;
```

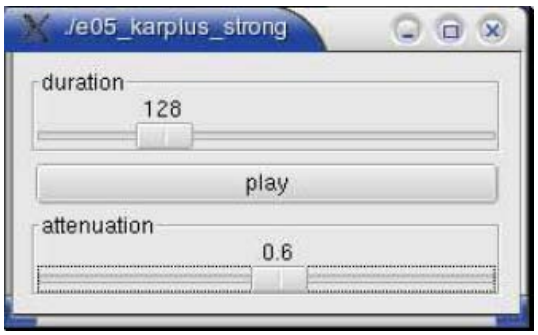

Figure 3: Screenshot of the Karplus-Strong example generated with the `jack-gtk.cpp` architecture

3 Faust and Q

Faust is tailored to DSP programming, and as such it is not a general-purpose programming language. In particular, it does not by itself have any facilities for other tasks typically encountered in signal processing and synthesis programs, such as accessing the operating system environment, real-time processing of audio and MIDI data, or presenting a user interface for the application. Thus, as we already discussed in the preceding section, all Faust-generated DSP programs need a supporting infrastructure (embodied in the architecture file) which provides those bits and pieces.

One of the architectures included in the Faust distribution is the Q language interface. Q is an interpreted functional programming language which has the necessary facilities for doing general-purpose programming as well as soft real-time processing of MIDI, OSC a.k.a. Open Sound Control (Wright et al., 2003) and audio data. The Q-Faust interface allows Faust DSPs to be loaded from a Q script at runtime. From the perspective of the Faust DSP, Q acts as a programmable supporting environment in which it operates, whereas in Q land, the DSP module is used as a "blackbox" to which the script feeds chunks of audio and control data, and from which it reads the resulting audio output. By these means, Q and Faust programs can be combined in a very flexible manner to implement full-featured software synthesizers and other DSP applications.

In this section we give a brief overview of the Q-Faust interface, including a simple but complete monophonic synthesizer example. For lack of space, we cannot give an introduction to the Q language here, so instead we refer the reader to (Gräf, 2005) and the extensive documentation available on the Q website at `http://q-lang.sf.net`.

3.1 Q module architecture

Faust's side of the Q-Faust interface consists of the *Q architecture file*, a little C++ code template `q.cpp` which is used with the Faust compiler to turn Faust DSPs into shared modules which can be loaded by the Q-Faust module at runtime. This file should already be included in all recent Faust releases, otherwise you can also find a copy of the file in the Q-Faust distribution tarball.

Once the necessary software has been installed, you should be able to compile a Faust DSP to a shared module loadable by Q-Faust as follows:

```
$ faust -a q.cpp -o mydsp.cpp mydsp.dsp
$ g++ -shared -o mydsp.so mydsp.cpp
```

Note: If you want to load several different DSPs in the same Q script, you have to make sure that they all use distinct names for the `mydsp` class. With the current Faust version this can be achieved most easily by just redefining

mydsp, to whatever class name you choose, during the C++ compile stage, like so:

```
$ g++ -shared -Dmydsp=myclassname
   -o mydsp.so mydsp.cpp
```

3.2 The Q-Faust module

The compiled DSP is now ready to be used in the Q interpreter. A minimal Q script which just loads the DSP and assigns it to a global variable looks as follows:

```
import faust;
def DSP = faust_init "mydsp" 48000;
```

The first line of the script imports Q's faust module which provides the operations to instantiate and operate Faust DSPs. The faust_init function loads a shared module (mydsp.so in this example, the .so suffix is supplied automatically) and returns an object of Q type FaustDSP which can then be used in subsequent operations. The second parameter of faust_init, 48000 in this example, denotes the sample rate in Hz. This can be an arbitrary integer value which is available to the hosted DSP (it is up to the DSP whether it actually uses this value in some way).

In the following examples we assume that you have actually loaded the above script in the Q interpreter; the commands below can then be tried at the interpreter's command prompt.

The faust_info function can be used to determine the number of input/output channels as well as the "UI" (a data structure describing the available control variables) of the loaded DSP:

```
==> def (N,M,UI) = faust_info DSP
```

To actually run the DSP, you'll need some audio data, encoded using 32 bit (i.e., single precision) floating point values as a byte string. (A *byte string* is a special kind of data object which is used in Q to represent arbitrary binary data, such as a C vector with audio samples in this case.) Suppose you already have two channels of audio data in the IN1 and IN2 variables and the DSP has 2 input channels, then you would pass the data through the DSP as follows:

```
==> faust_compute DSP [IN1,IN2]
```

This will return another list of byte strings, containing the 32 bit float samples produced by the DSP on its output channels, being fed with the given input data.

Some DSPs (e.g., synthesizers) don't actually take any audio input, in this case you just specify the number of samples to be generated instead:

```
==> faust_compute DSP 1024
```

Most DSPs also take additional control input. The control variables are listed in the UI component of the faust_info return value. For instance, suppose that there is a "Gain" parameter listed there, it might look as follows:

```
==> controls UI!0
hslider <<Ref>> ("Gain",1.0,0.0,10.0,0.1)
```

The second parameter of the hslider constructor indicates the arguments the control was created with in the .dsp source file (see the Faust documentation for more details on this). The first parameter is a Q *reference* object which points to the current value of the control variable. The reference can be extracted from the control description with the control_ref function and you can then change the value with Q's put function before invoking faust_compute (changes of control variables only take effect between different invokations of faust_compute):

```
==> def GAIN = control_ref (controls UI!0)

==> put GAIN 2.0
```

3.3 Monophonic synthesizer example

For a very simple, but quite typical and fully functional example, let us take a look at the monophonic synthesizer program in Figure 4. It basically consists of two real-time threads: a control loop which takes MIDI input and changes the synth DSP's control variables accordingly, and an audio loop which just pulls audio data from the DSP at regular intervals and outputs it to the audio interface. The Faust DSP we use here is the simple additive synth shown in Figure 5.

The header section of the Q script imports the necessary Q modules and defines some global variables which are used to access the MIDI input and audio output devices as well as the Faust DSP. It also extracts the control variables from the Faust DSP and stores them in a dictionary, so that we can finally assign the references to a corresponding collection of global variables. These variables are then used in the control loop to set the values of the control variables.

```
import audio, faust, midi;

def (_,_,_,_,SR) = audio_devices!AUDIO_OUT,
  SR = round SR, BUFSZ = 256,
  IN = midi_open "Synth",
  _ = midi_connect (midi_client_ref
                    "MidiShare/ALSA Bridge") IN,
  OUT = open_audio_stream AUDIO_OUT PA_WRITE
        (SR,1,PA_FLOAT32,BUFSZ),
  SYNTH = faust_init "synth" SR,
  (N,M,UI) = faust_info SYNTH, CTLS = controls UI,
  CTLD = dict (zip (map control_label CTLS)
                   (map control_ref CTLS));

def [FREQ,GAIN,GATE] =
  map (CTLD!) ["freq","gain","gate"];

/***********************************************/

freq N     = 440*2^((N-69)/12);
gain V     = V/127;

process (_,_,_,note_on _ N V)
           = put FREQ (freq N) ||
             put GAIN (gain V) ||
             put GATE 1 if V>0;
           = put GATE 0 if freq N = get FREQ;

midi_loop  = process (midi_get IN) || midi_loop;

audio_loop = write_audio_stream OUT
             (faust_compute SYNTH BUFSZ!0) ||
             audio_loop;

/***********************************************/

def POL = SCHED_RR, PRIO = 10;
realtime   = setsched this_thread POL PRIO;

synth      = writes "Hit <CR> to stop: " ||
             reads || ()
  where H1 = thread (realtime || midi_loop),
    H2 = thread (realtime || audio_loop);
```

Figure 4: Q script for the monophonic synth example

The second section of the code contains the definitions of the control and audio loop functions. It starts out with two helper functions `freq` and `gain` which are used to map MIDI note numbers and velocities to the corresponding frequency and gain values. The `process` function (not to be confused with the `process` "main" function of the Faust program!) does the grunt work of translating an incoming MIDI event to the corresponding control settings. In this simple example it does nothing more than responding to note on and off messages (as usual, a note off is just a note on with velocity 0). The example also illustrates how MIDI

```
import("music.lib");

// control variables

vol  = nentry("vol", 0.3, 0, 10, 0.01);

attk = nentry("attack", 0.01, 0, 1, 0.001);
decy = nentry("decay", 0.3, 0, 1, 0.001);
sust = nentry("sustain", 0.5, 0, 1, 0.01);
rels = nentry("release", 0.2, 0, 1, 0.001);

freq = nentry("freq", 440, 20, 20000, 1);
gain = nentry("gain", 1, 0, 10, 0.01);
gate = button("gate");

// simple monophonic synth

smooth(c) = *(1-c) : +~*(c);

voice   = gate : adsr(attk, decy, sust, rels) :
          *(osci(freq)+0.5*osci(2*freq)+
            0.25*osci(3*freq)) :
          *(gain : smooth(0.999));

process = vgroup("synth", voice : *(vol));
```

Figure 5: Faust source for the monophonic synth example

messages are represented as an "algebraic" data type in Q, and how the note and velocity information is extracted from this data using "pattern matching." In the case of a note on message we change the `FREQ` and `GAIN` of the single synth voice accordingly and then set the `GATE` variable to 1, to indicate that a note is playing. For a note off message, we simply reset the `GATE` variable to 0; in the DSP, this triggers the release phase of the synth's ADSR envelop.

The `process` function is invoked repeatedly during execution of `midi_loop`. The `audio_loop` function just keeps reading the audio output of the DSP and sends it to the audio output stream. The two loops are to be executed asynchronously, in parallel. (It is worth noting here that the necessary protection of shared data, i.e., the control variable references, is done automatically behind the scenes.)

The third section of the script contains the main entry point, the `synth` function which kicks off two real-time threads running the `midi_loop` and `audio_loop` functions and then waits for user input. The function returns a "void" `()` value as soon as the user hits the carriage return key. (At this point the two thread handles `H1` and `H2` are garbage-collected immediately and the corresponding threads are thus terminated automatically, so there is no need to

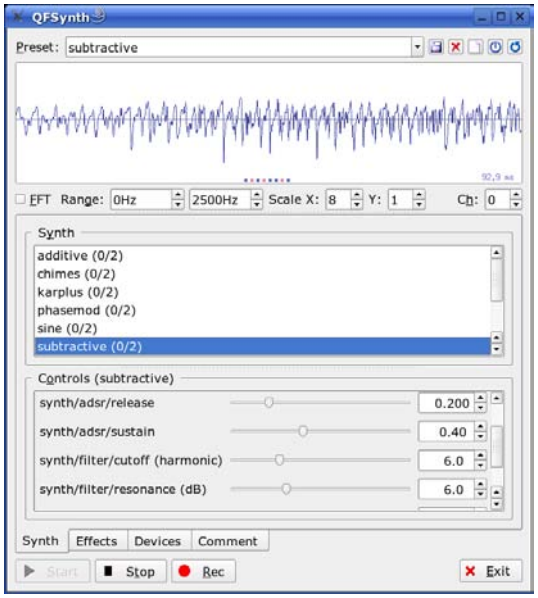

Figure 6: QFSynth program

explicitly cancel the threads.)

Of course the above example is rather limited in functionality (that shouldn't come as a big surprise as it is just about one page of Faust and Q source code). A complete example of a Faust-based polyphonic software synthesizer with GUI can be found in the QFSynth application (cf. Figure 6) which is available as a separate package from the Q website.

3.4 Q, Faust and SuperCollider

The Q-Faust interface provides a direct way to embed Faust DSPs in Q programs, which is useful for testing DSPs and for simple applications with moderate latency requirements. For more elaborate applications it is often convenient to employ a dedicated software synthesis engine which does the grunt work of low-latency control data and audio processing. This is where Q's OSC-based SuperCollider interface (Gräf, 2005) comes in handy. Using SuperCollider's Faust plugin interface, described in the next section, Faust DSPs can also be loaded into the SuperCollider sound server and are then ready to be operated from Q programs via OSC.

4 Faust and SuperCollider3

SuperCollider3 (McCartney, 2002) is a realtime synthesis and composition framework, divided into a synthesis server application (`scsynth`) and an object-oriented realtime language (`sclang`). Any application capable of sending OpenSoundControl (Wright et al., 2003) messages can control `scsynth`, one notable example being Q (section 3).

Correspondingly, support for plugins generated by Faust is divided into an interface to `scsynth` and `sclang`, respectively.

4.1 Interface to `scsynth`

In order to compile a Faust plugin for the SuperCollider3 synthesis architecture, you have to use the corresponding architecture file:

```
$ faust -a supercollider.cpp \
    -o noise.cpp noise.dsp
```

For compiling the plugin on Linux you can use the provided `pkg-config` specification, which is installed automatically when you pass `DEVELOPMENT=yes` to `scons` when building SuperCollider:

```
$ g++ -shared -o noise.so \
    `pkg-config --cflags libscsynth` \
    noise.cpp
```

The resulting plugin should be put in a place where `scsynth` can find it, e.g. into `~/share/SuperCollider/Extensions/Faust` on Linux.

Unit-generator plugins in SuperCollider are referenced by name on the server; the plugin generated by Faust currently registers itself with the C++ filename sans extension. In future versions of Faust the plugin name will be definable in the process specification itself.

4.2 Interface to `sclang`

Faust can produce an XML description of a plugin, including various meta data and the structural layout of the user interface.

This information is used by `faust2sc` in the Faust distribution to generate a SuperCollider class file, which can be compiled and subsequently used from within `sclang`.

For example,

```
$ faust -xml -o /dev/null noise.dsp
$ faust -xml -o /dev/null karplus.dsp
$ faust2sc -p Faust -o Faust.sc \
    noise.dsp.xml karplus.dsp.xml
```

generates a SuperCollider source file, that, when compiled by sclang, makes available the respective plugins for use in synth definitions.

Now copy the source file into `sclang`'s search path, e.g. `~/share/SuperCollider/Extensions/Faust` on Linux.

Since `scsynth` doesn't provide GUI facilities, UI elements in Faust specifications are mapped

to control rate signals on the synthesis server. The argument order is determined by the order of appearance in the (flattened) block diagram specification; audio inputs (named in1 ... inN) are expected before control inputs. The freeverb example plugin has the following arguments to the `ar` instance creation method when used from `sclang`:

```
in1 in2 damp(0.5) roomsize(0.5) wet(0.3333)
```

i.e. first the stereo input pair followed by the control inputs including default values.

4.3 Examples

Unsurprisingly plugins generated by Faust can be used just like any other unit generator plugin, although the argument naming can be a bit verbose, depending on the labels used in UI definitions.

Assuming the server has been booted, the "noise" example found in the distribution can be tested like this:

```
{ Pan2.ar(
  FaustNoise.ar(0.2),
  LFTri.kr(0.1) * 0.4)
}.play
```

A more elaborate example involves the "karplus" example plugin and shows how to use keyword arguments.

```
{
  FaustKarplus.ar(
    play: { |i|
      Impulse.kr(
        exprand(10/6*(i+1), 20)
        * SinOsc.kr(0.1).range(0.3, 1)
      )
    } ! 6,
    duration_samples: LFSaw.kr(0.1)
                      .range(80, 128),
    attenuation: LFPar.kr(0.055, pi/2)
                 .range(0.1, 0.4)
                 .squared,
    level: 0.05
  ).clump(2).sum
}.play
```

Note that the *trigger* button in the *jack-gkt* example has been replaced by a control rate impulse generator connected to the *play* input.

Rewriting the monophonic synth example from section 3.3 in SuperCollider is a matter of recompiling the plugin,

```
$ faust -a supercollider.cpp \
    -o synth.cpp synth.dsp
$ g++ -shared -o synth.so \
    `pkg-config --cflags libscsynth` \
    synth.cpp
$ faust -xml -o /dev/null synth.dsp
$ faust2sc -p Faust -o FaustSynth.sc \
    synth.dsp.xml
```

and installing `synth.so` and `FaustSynth.sc` to the appropriate places.

The corresponding SynthDef just wraps the Faust plugin:

```
(
SynthDef(\faustSynth, {
  | trig(0), freq(440), gain(1),
    attack(0.01), decay(0.3),
    sustain(0.5), release(0.2) |
  Out.ar(
    0,
    FaustSynth.ar(
      gate: trig,
      freq: freq,
      gain: gain,
      attack: attack,
      decay: decay,
      sustain: sustain,
      release: release
    )
  )
}, [\tr]).send(s)
)
```

and can now be used with SuperCollider's pattern system:

```
(
TempoClock.default.tempo_(2);
x = Synth(\faustSynth);
p = Pbind(
  \instrument, \faustSynth,
  \trig, 1,
  \sustain, 0.2,
  \decay, 0.1,
  \scale, #[0, 3, 5, 7, 10],
  \release, Pseq(
    [Pgeom(0.2, 1.5, 4),
     4,
     Pgeom(0.2, 0.5, 4)],
    inf
  ),
  \dur, Pseq(
    [Pn(1/4, 4),
     15.5/4,
     Pn(1/8, 4)],
    inf
  ),
  \degree, Pseq(
    [1, 2, 3, 4, 5, 2, 3, 4, 5].mirror,
    inf
  )
).play(
```

```
  protoEvent: (
    type: \set,
    args: [\trig, \freq, \release]
  )
)
)
```

5 Conclusion

Existing functional programming environments have traditionally been focused on non real-time applications such as artificial intelligence, programming language compilers and interpreters, and theorem provers. While multimedia has been recognized as one of the key areas which could benefit from functional programming techniques (Hudak, 2000), the available tools are not capable of supporting real-time execution with low latency requirements. This is unfortunate since real time is where the real fun is in multimedia applications.

The Faust programming language changes this situation. You no longer have to program your basic DSP modules in C or C++, which is a tedious and error-prone task. Faust allows you to develop DSPs in a high-level functional programming language which can compete with, or even surpass the efficiency of carefully hand-coded C routines. The SuperCollider Faust plugin interface lets you execute these components in a state-of-the-art synthesis engine. Moreover, using Q's Faust and SuperCollider interfaces you can also program the real-time control of multimedia applications in a modern-style functional programming language. Together, Faust, Q and SuperCollider thus provide an advanced toolset for programming DSP and computer music applications which should be useful both for practical application development and educational purposes.

References

Albert Gräf. 2005. Q: A functional programming language for multimedia applications. In *Proceedings of the 3rd International Linux Audio Conference (LAC05)*, pages 21–28, Karlsruhe. ZKM.

Paul Hudak. 2000. *The Haskell School of Expression: Learning Functional Programming Through Multimedia*. Cambridge University Press.

K. Karplus and A. Strong. 1983. Digital synthesis of plucked-string and drum timbres. *Computer Music Journal*, 7(2):43–55.

James McCartney. 2002. Rethinking the computer music language: SuperCollider. *Computer Music Journal*, 26(4):61–68. See also `http://supercollider.sourceforge.net`.

Y. Orlarey, D. Fober, and S. Letz. 2004. Syntactical and semantical aspects of Faust. *Soft Computing*, 8(9):623–632.

Matthew Wright, Adrian Freed, and Ali Momeni. 2003. OpenSound Control: State of the art 2003. In *Proceedings of the Conference on New Interfaces for Musical Expression (NIME-03)*, pages 153–159, Montreal.