FUNCTIONAL SIGNAL PROCESSING WITH PURE AND FAUST USING THE LLVM TOOLKIT

Albert Gräf
Dept. of Computer Music, Institute of Musicology
Johannes Gutenberg University Mainz
Dr.Graef@t-online.de

ABSTRACT

Pure and Faust are two functional programming languages useful for programming computer music and other multimedia applications. Faust is a domain-specific language specifically designed for synchronous signal processing, while Pure is a general-purpose language which aims to facilitate symbolic processing of complicated data structures in a variety of application areas. Pure is based on the LLVM compiler framework which supports both static and dynamic compilation and linking.

This paper discusses a new LLVM bitcode interface between Faust and Pure which allows direct linkage of Pure code with Faust programs, as well as inlining of Faust code in Pure scripts. The interface makes it much easier to integrate signal processing components written in Faust with the symbolic processing and metaprogramming capabilities provided by the Pure language. It also opens new possibilities to leverage Pure and its JIT (just-in-time) compiler as an interactive frontend for Faust programming.

1. INTRODUCTION

Programming signal processing applications in imperative programming languages can be a difficult and error-prone task. Functional programming provides an alternative way to tackle these problems. It allows signals to be modelled either as discrete functions of time or, equivalently, as streams (potentially infinite lists) of samples or control messages. The "patching" of signal processing components can then be expressed very conveniently through the combination of higher-order functions. An important benefit of this approach is that functional programs typically have simpler semantics and can thus serve as formal and platform-independent specifications of signal processing components.

Pure and Faust are two functional programming languages which are useful in this context and which nicely complement each other. Faust is a statically typed language based on the lambda calculus which has been designed specifically for synchronous processing of numeric signals at the sample level [1]. It enables you to build complicated signal processors from simple components such as pointwise arithmetic operations and delays by means of its built-in block diagram algebra. Faust has an optimizing compiler with which Faust programs can be compiled to efficient native code.

In contrast, Pure is a dynamically typed general-purpose language tailored for symbolic processing, which can be used to tackle the higher-level components of computer music and other multimedia applications [2]. Pure is based on term rewriting, thus Pure programs are essentially collections of symbolic equations which are used to evaluate expressions by reducing them to their simplest form. Pure compiles the term rewriting rules to efficient native code so that they are executed very efficiently. While Pure doesn't specifically target numeric signal processing, it provides a matrix data structure akin to MATLAB/Octave which can be used to handle copious amounts of sample data and interface to numeric signal processing code written in other languages in an efficient way.

While Faust is batch-compiled, Pure has a just-in-time (JIT) compiler and is typically used in an interactive fashion, either as a standalone programming environment or as an embedded scripting language in other environments such as Miller Puckette's Pd. Pure is based on the LLVM compiler toolkit, which opens some interesting possibilities to interface it with other LLVM-capable languages.

LLVM, the "Low-Level Virtual Machine", is an open-source cross-platform compiler backend available under a BSD-style license, which forms the backbone of a number of important compiler projects, including Apple's llvm-gcc and the new clang compiler [3]. It is also used in Google's "UnladenSwallow" Python compiler [4] and the latest versions of the Glasgow Haskell compiler [5], as well as in OpenCL implementations by Apple, AMD and NVIDIA [6]. Besides static compilation, LLVM also offers JIT compilation which makes it usable in dynamic environments where bits of source code are compiled at runtime as needed before being executed. This is also the way it is typically used in Pure, although the Pure interpreter can also be invoked as a static compiler in order to produce native executables and libraries.

LLVM exposes a fairly low-level code model (somewhere between real assembler and C) to client frontends. This makes it a useful target for signal processing languages where the generation of efficient output code is very important. Thus an LLVM backend has been on the wishlist of Faust developers and users alike for some time. Such a backend is now available [7]. This paper reports on sub-

sequent work by the author to build an LLVM-based bridge between Faust and Pure.

Note that Faust requires a considerable amount of "glue code" to work in different environments, which is usually provided in the form of special C++ modules (known as "architectures" in Faust parlance). The Pure-Faust bridge described here allows this glue code to be written in Pure instead, which has the advantage that Faust modules can be tested and run interactively in a dynamic, interpreter-like environment without sacrificing execution speed.

2. USING FAUST WITH LLVM

To take advantage of Faust's new LLVM backend, you currently need a fairly recent snapshot of the "faust2" branch of the compiler in the Faust git repository [7].

We'll use the following little Faust module as a running example throughout this paper. It implements a simple additive synthesizer with control variables `gain` (volume), `gate` (note on/off) and `freq` (fundamental frequency in Hz). The output is the sum of three sine oscillators for the fundamental and the first and second overtone. [1]

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

freq = nentry("freq", 440, 20, 20000, 1);
gain = nentry("gain", 0.3, 0, 10, 0.01);
gate = button("gate");

amp(1) = 1.0; amp(2) = 0.5; amp(3) = 0.25;
partial(i) = amp(i+1)*osc((i+1)*freq);

process = sum(i, 3, partial(i)) * gain
  * (gate : adsr(0.01, 0.3, 0.5, 0.2));
```

The `-lang llvm` option instructs the Faust compiler to output LLVM bitcode (instead of the usual C++ code). Also, for using Faust-generated code with Pure, you want to add the `-double` option to make the compiled Faust module use double precision floating point values for samples and control values. [2] So, if you saved the above Faust code in a source file `organ.dsp`, say, you'd compile this module as follows:

```
faust -double -lang llvm organ.dsp -o organ.bc
```

If you did everything right, you should now have the LLVM bitcode for our little Faust organ in the `organ.bc` file which is ready to be loaded by the Pure interpreter, as described in the following section. If you want, you can also have the Faust compiler print the code in a human-readable format (LLVM assembler) by omitting the `-o organ.bc` option. A description of this format can be found on the LLVM website [3].

3. LOADING A BITCODE MODULE IN PURE

The Pure interpreter has the capability to load arbitrary LLVM bitcode modules and make the external functions defined in that code callable from Pure. It is worth mentioning here that the ability to load Faust modules is in fact just a special instance of this facility. Pure can import and inline code written in a number of different programming languages supported by LLVM-capable compilers (C, C++ and Fortran at present), but in the following we concentrate on the Faust bitcode loader which has special knowledge about the Faust language built into it.

Loading a Faust bitcode module in Pure is done with a special kind of import clause which looks as follows (assuming that you have compiled the `organ.dsp` example from the previous section beforehand):

```
using "dsp:organ";
```

It's not necessary to supply the `.bc` bitcode extension, it will be added automatically. You can repeat this statement as often as you want; the bitcode loader then checks whether the module has changed (i.e., was recompiled since it was last loaded) and reloads it if necessary. On the Pure side, the callable functions of the Faust module look as shown in Figure 1. (This uses pretty much the same syntax as C extern declarations; you can obtain this listing yourself by typing `show -g organ::*` in the Pure interpreter after loading the module.) Also note that the interpreter automatically places the interface functions of the Faust module in their own `organ` namespace in order to avoid name clashes if several different Faust modules are loaded in the same Pure program.

The most important interface routines are `new`, `init` and `delete` (used to create, initialize and destroy an instance of the dsp) and `compute` (used to apply the dsp to a given block of samples). Two useful convenience functions are added by the Pure compiler: `newinit` (which combines `new` and `init`) and `info`, which yields pertinent information about the dsp as a Pure tuple containing the number of input and output channels and the Faust control descriptions. The latter are provided in a symbolic format ready to be used in Pure; more about that in the Section 5. Also note that there's usually no need to explicitly invoke the `delete` routine in Pure programs; the Pure compiler makes sure that this routine is added automatically as a finalizer to all dsp pointers created through the `new` and `newinit` routines so that dsp instances are destroyed automatically when the corresponding Pure objects are garbage-collected.

4. INLINING FAUST CODE

Instead of compiling a Faust module manually and loading the resulting bitcode module in Pure, you can also just inline Faust programs directly in Pure. The necessary steps to compile and load the module will then be handled automatically by the Pure interpreter. To do this, you just enclose the Faust code in Pure's inline code brackets. Behind the opening bracket, there's a special tag identifying the contents as Faust source, which takes the form '`-*- dsp:`*name* `-*-`'. The '`dsp`' tag tells the compiler that what follows is Faust code, while the given *name* indicates the name of the Faust module (which, as we've seen, becomes the namespace into which the Pure compiler places

[1] To keep things simple, we have hardcoded the relative amplitudes of the partials and the parameters of the ADSR envelop here. In a real application, you'd probably want to turn these into additional control variables.

[2] The `-double` option isn't strictly necessary, but it makes interfacing between Pure and Faust easier and more efficient, since the Pure interpreter uses `double` as its native floating point format. This option is also added automatically when inlining Faust code (see Section 4).

```
extern void buildUserInterface(struct_dsp_llvm*, struct_UIGlue*) = organ::buildUserInterface;
extern void classInit(int) = organ::classInit;
extern void compute(struct_dsp_llvm*, int, double**, double**) = organ::compute;
extern void delete(struct_dsp_llvm*) = organ::delete;
extern void destroy(struct_dsp_llvm*) = organ::destroy;
extern int getNumInputs(struct_dsp_llvm*) = organ::getNumInputs;
extern int getNumOutputs(struct_dsp_llvm*) = organ::getNumOutputs;
extern expr* info(struct_dsp_llvm*) = organ::info;
extern void init(struct_dsp_llvm*, int) = organ::init;
extern void instanceInit(struct_dsp_llvm*, int) = organ::instanceInit;
extern struct_dsp_llvm* new() = organ::new;
extern struct_dsp_llvm* newinit(int) = organ::newinit;
```

Figure 1. Call interfaces for the sample Faust module on the Pure side.

the Faust interface routines). The inline code section for our previous example would thus look as follows: [3]

```
%< -*- dsp:organ -*-
import("music.lib");

freq = nentry("freq", 440, 20, 20000, 1);
gain = nentry("gain", 0.3, 0, 10, 0.01);
gate = button("gate");

amp(1) = 1.0; amp(2) = 0.5; amp(3) = 0.25;
partial(i) = amp(i+1)*osc((i+1)*freq);

process = sum(i, 3, partial(i)) * gain
  * (gate : adsr(0.01, 0.3, 0.5, 0.2));
%>
```

You can insert these lines into a Pure script, or just type them directly at the prompt of the Pure interpreter. This method is particularly convenient when experimenting with small Faust modules interactively in the Pure interpreter, as it eliminates the edit-compile-link cycle needed when compiling the Faust modules separately.

5. RUNNING A FAUST DSP IN PURE

Let us now take a look at how we can run the Faust organ in Pure to generate some samples. This process generally involves the steps sketched out below. After loading (or inlining) the Faust module, you can type these commands at the command prompt ('> ') of the Pure interpreter.

Step 1. We first create an instance of the Faust signal processor using the `newinit` routine, and assign it to a Pure variable as follows:

```
> let dsp = organ::newinit 44100;
```

Note that the constant `44100` denotes the desired sample rate in Hz. This can be an arbitrary integer value, which is available in the Faust program by means of the `SR` variable.

Step 2. The dsp is now fully initialized and we can use it to compute some samples. But before we can do this, we'll generally need to know how many channels of audio data the dsp consumes and produces, and which control variables it provides. This information can be extracted with the `info` function, and be assigned to some Pure variables as follows:

```
> let k,l,ui = organ::info dsp;
> k,l;
0,1
```

Note that in our example, there's no audio input and just one channel of output samples (i.e., a mono output signal). We'll have a look at the control variables later.

Step 3. Next we'll need to prepare input and output buffers to hold the samples passed to and computed by the Faust module. Pure's Faust interface allows us to pass Pure double matrices as sample buffers, which makes this step quite convenient. In our example, we need a $k \times n$ matrix (which is an empty matrix in this case) for the input and a $l \times n$ matrix for the output. Here, n is the desired *block size* (the number of samples to be processed in one go). That is, there's one row in the matrices for each audio channel, and the size of each row is the block size. Suitable matrices can be created with appropriate calls of the `dmatrix` function defined in Pure's standard library:

```
> let n = 10; // the block size
> let in = dmatrix (k,n);
> let out = dmatrix (l,n);
> in; out;
{}
{0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0}
```

Step 4. We can now apply the dsp by invoking its `compute` routine:

```
> organ::compute dsp n in out, out;
{0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0}
```

So the output is still all zeros, which is hardly surprising since we didn't switch on the `gate` control yet. Information about the available controls can be found in the `ui` variable which we defined in step 2 above:

```
> ui;
vgroup ("organ",
[nentry #<pointer 0xf55ef8>
 ("freq",440.0,20.0,20000.0,1.0),
 nentry #<pointer 0xf55eb8>
 ("gain",0.3,0.0,10.0,0.01),
 button #<pointer 0xf55ec0> "gate"])
```

[3] The following Pure code requires Pure 0.47 or later, available on the Pure website at `http://pure-lang.googlecode.com`. If you get a syntax error trying to input the inline code sections, then most likely you have an older Pure version installed.

In general, this data structure takes the form of a tree which corresponds to the hierarchical layout of the control groups and values in the Faust program. In this case, we just have one toplevel group named 'organ' which is created by the Faust compiler. This group contains the freq, gain and gate parameters of the Faust dsp, which are represented in Pure as a list containing the relevant information about the type and name of each control, along with a double pointer which can be used to inspect and modify the control value. Where applicable, the data structure also lists the initial value, range and step size of a control, as they were specified in the Faust source.

While it's possible to access this information in a direct fashion, there's also a faustui.pure module included in the Pure distribution which makes this easier. First we extract the mapping of control variable names to the corresponding double pointers as follows:

```
> using faustui;
> let ui = control_map (controls ui); ui;
{"freq"=>#<pointer 0xf55ef8>,
"gain"=>#<pointer 0xf55eb8>,
"gate"=>#<pointer 0xf55ec0>}
```

The result is a Pure record value indexed by control names. E.g., the pointer which belongs to our gate control can be obtained with ui!"gate" (note that '!' is Pure's indexing operator). There are also convenience functions to inspect a control and change its value. Let's see what happens if we set the gate control to 1.0 (meaning "on"):

```
> let gate = ui!"gate";
> get_control gate;
0.0
> put_control gate 1.0;
()
> get_control gate;
1.0
> organ::compute dsp n in out, out;
{0.000916099235344598,0.00187158614672218,
0.00283108045199002,0.00376191957462151,
0.00463511838936804,0.00542654633209689,
0.00611762817140048,0.00669550599165544,
0.00715357741464203,0.00749105048578719}
> organ::compute dsp n in out, out;
{0.00771236488124018,0.00782759103035606,
0.00785038625400857,0.00779836818632883,
0.00768982665928043,0.0075451428235108,
0.00738392801978776,0.00722516057726323,
0.0070835316007035,0.00697180757718093}
```

So we finally got some real output now. Note that the compute routine also modifies the internal state of the dsp instance so that a subsequent call will continue with the output stream where the previous call left off. Thus we can now just keep on calling compute to compute as much of the output signal as we need.

6. EXAMPLE: A PD OBJECT

After walking through the computation step by step, it is now an easy matter to turn this into a working program. Figure 2 shows the complete Pure code for an object named organ~ ready to be loaded in Pd. To make this work, you need the pd-pure plugin loader (available as an addon module from the Pure website) which equips Pd with the capability to run external objects written in Pure [8]. A sample patch showing this object in action can be seen in Figure 3.

Note that the organ_dsp function of the program is the main entry point exposed to Pd which does all the necessary interfacing to Pd. Besides the audio processing itself, this also includes setting the control parameters of the Faust dsp in response to incoming "note" messages.

The object is actually implemented by a local function organ which carries with it the required information (the dsp instance, number of input and output channels, the control variables and the output buffer) as local variables defined by the **when** clause; this makes it possible to have several different organ~ objects in the same Pd patch. The organ_dsp function returns this local function along with the number of audio inputs and outputs so that Pd can properly set up the object when it is inserted into a patch.

Note that the organ function is invoked by Pd with either a matrix or a control message as argument. The former happens when the object is run by Pd's audio processing loop in order to produce a block of samples; the organ function handles this by simply invoking the Faust dsp and returning the output buffer to Pd. The latter case occurs whenever the object receives a control message. In this example, we have added code to handle lists of MIDI note numbers and velocities, which get translated to the appropriate settings of the control variables of the Faust dsp.

Also note that this implementation uses an "actor style" of processing which is close to how Pd works but involves local state. There are other ways to do this in a more functional style by using streams, see [2] for details.

Finally, note that instead of importing the Faust module with a **using** clause, we might just as well have inlined the Faust code as described in Section 4. By using the interactive live editing facilities provided by pd-pure, it then becomes possible to change both the Faust module and the control processing in the Pure part of the code on the fly, while the Pd patch keeps running. For details on these "livecoding" facilities we refer the reader to the pd-pure documentation [8].

7. CONCLUSION

The facilities described in this paper are fully implemented in the latest versions of the Pure and Faust compilers. They enable programmers to employ Pure as an alternative hosting environment for Faust. Programming the required glue code for interfacing Faust to other environments in Pure offers some substantial advantages over the C++ architecture interface supplied by the Faust compiler. Specifically, Pure provides a convenient interactive environment which facilitates testing and livecoding of Faust components. Pure scripts containing Faust code can also be batch-compiled to native executables in order to implement efficient standalone applications. In either case, you can use Pure's collection of addon modules for pre- and postprocessing Faust input and output signals in any desired manner. In particular, Pure interfaces nicely with the GNU Scientific Library, Octave and Gnumeric, and it also offers the necessary facilities to deal with MIDI, OSC, audio and graphics in a

```
// organ~.pure

// These are provided by the Pd runtime.
extern float sys_getsr(), int sys_getblksize();

// Get Pd's default sample rate and block size.
const SR = int sys_getsr;
const n = sys_getblksize;

// Load the dsp.
using "dsp:organ";
using faustui;

organ_dsp = k,l,organ with
  // The dsp loop.
  organ in::matrix = organ::compute dsp n in out $$ out;
  // Respond to note messages.
  organ [num, vel] = put_control freq (midicps num) $$ // note on
    put_control gain (vel/127) $$ put_control gate 1.0 if vel>0;
  = put_control gate 0.0 otherwise; // note off
  // Translate MIDI note numbers to frequencies in Hz.
  midicps num = 440*2^((num-69)/12);
end when
  // Initialize an instance of the dsp.
  dsp = organ::newinit SR;
  // Get the number of inputs and outputs and the control variables.
  k,l,ui = organ::info dsp;
  ui = control_map (controls ui);
  {freq,gain,gate} = ui!!["freq","gain","gate"];
  // Create a buffer large enough to hold the output from the dsp.
  out = dmatrix (l,n);
end;
```

Figure 2. Pure code for organ object to be loaded in Pd.

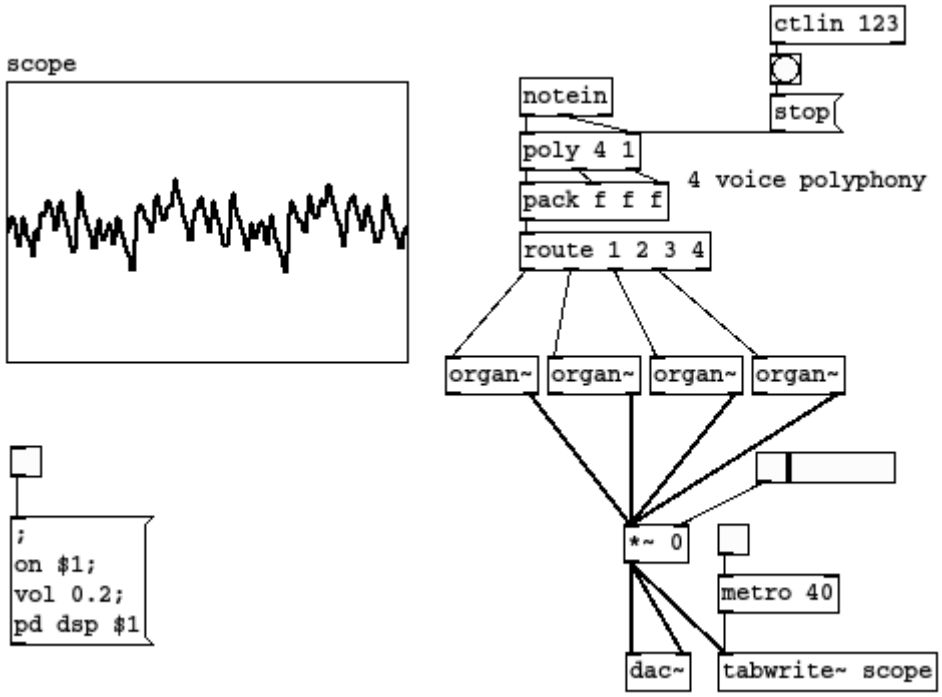

Figure 3. Organ objects in a Pd patch.

direct manner. GUI, database and web programming interfaces are provided, too.

Thus using Pure with Faust provides a great deal of flexibility and interactivity, more than can be achieved by using Faust modules directly in Csound, Pd, SuperCollider and similar environments, which don't provide facilities to change Faust code on the fly (at least not yet), and by design aren't general-purpose programming environments either. On the other hand, these are dedicated audio environments which have been designed specifically for (hard) realtime applications. In contrast, Pure only provides soft (i.e., best effort) realtime capabilities. We have found, however, that since Pure is compiled to native code, it is usually able to keep up with realtime loops quite easily, as long as the block size is sufficiently large and one doesn't try to squeeze too much computation into the realtime loop. E.g., Pd's realtime loop runs at 64 samples by default (offering latencies in the 1 msec ballpark) and Pure code running inside that loop usually works just fine, even if it is considerably more complicated than the simple example shown in this paper.

An interesting avenue for further research is to employ Pure as an interactive frontend to Faust. This is now possible (and in fact quite easy), since Pure allows Faust source to be created under program control and then compiled on the fly using Pure's built-in `eval` function. Taking this idea further, one might leverage Pure's symbolic computation capabilities in order to embed Faust as a domain-specific sublanguage in Pure. This should provide an interesting alternative to other interactive signal processing environments based on Lisp dialects such as Snd-Rt [9].

Acknowledgments

Many thanks go to Stéphane Letz at Grame for his work on the Faust LLVM interface which made this project possible in the first place. Special thanks are also due to Yann Orlarey for inviting me to Grame to work on improving our arsenal of functional signal processing tools.

8. REFERENCES

[1] Y. Orlarey, D. Fober, and S. Letz, "Syntactical and semantical aspects of Faust," *Soft Computing*, vol. 8, no. 9, pp. 623–632, 2004.

[2] A. Gräf, "Signal processing in the Pure programming language," in *Proceedings of the 7th International Linux Audio Conference*. Parma: Casa della Musica, 2009.

[3] C. Lattner et al, "The LLVM compiler infrastructure," http://llvm.org, 2011.

[4] "UnladenSwallow: a faster implementation of Python," http://unladen-swallow.googlecode.com, 2011.

[5] D. A. Terei and M. M. Chakravarty, "An LLVM backend for GHC," in *Proceedings of the third ACM SIGPLAN Haskell Symposium*, ser. Haskell '10. New York, NY, USA: ACM, 2010, pp. 109–120.

[6] "OpenCL: The open standard for parallel programming of heterogeneous systems," http://www.khronos.org/opencl, 2011.

[7] S. Letz, "LLVM backend for Faust," http://www.grame.fr/~letz/faust_llvm.html, 2011.

[8] A. Gräf, "pd-pure: Pd loader for Pure scripts," http://docs.pure-lang.googlecode.com/hg/pd-pure.html, 2011.

[9] K. Matheussen, "Realtime music programming using Snd-Rt," in *Proceedings of the International Conference on Computer Music*. Belfast: Queen's University, 2008.