An LLVM bitcode interface between Pure and Faust

Albert Gräf
Dept. of Computer Music, Institute of Musicology
Johannes Gutenberg University
55099 Mainz, Germany
Dr.Graef@t-online.de

Abstract

This paper discusses the 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 LLVM-based JIT (just-in-time) compiler as an interactive frontend for Faust programming.

Keywords

Functional programming, Faust, Pure, LLVM, signal processing.

1 Introduction

Pure and Faust are two functional programming languages which are useful in creating signal processing applications of various kinds. The two languages complement each other. While Faust is a statically typed domain-specific language for creating numeric signal processing components which work at the sample level [7], 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]. Both Pure and Faust have compilers producing native code; however, 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 Pd.

Faust has had a Pure plugin architecture for some time already. However, this has been somewhat awkward to use since the programmer always has to go through an edit-compile-link cycle in order to create a shared library object of the Faust plugin, which can then be loaded in Pure. The new LLVM bitcode interface makes this much easier.

LLVM, the "Low-Level Virtual Machine", is an open-source cross-platform compiler backend available under a BSD-style license [4], which forms the backbone of a number of important compiler projects, including Apple's latest incarnations of the GNU compiler collection as well as *clang*, a new C/C++ compiler featuring various improvements over gcc [1]. In the past few years, the LLVM project has attracted a number of compiler writers who are retargeting compilers and interpreters to use LLVM. Google's Python compiler "UnladenSwallow" [9] and David A. Terei's backend for the Glasgow Haskell Compiler [8] are just two notable examples. Pure has used LLVM as its backend since the very first Pure release in 2008.

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, and this backend was finally designed and implemented by Stéphane Letz at Grame in 2010. The new backend is now available in the "faust2" branch in Faust's git repository [5]. During a brief visit of the author at Grame last year, we started working on leveraging the LLVM support of Faust and Pure to build a better bridge between the two languages. This paper reports on the results of this cooperation.

In Sections 2 and 3 we first take a brief look at the Faust and Pure sides of the new Pure-Faust bridge, respectively, discussing Faust's LLVM backend and Pure's LLVM bitcode loader. In Section 4 we walk the reader through the steps required to run a Faust module in Pure. Section 5 explains how to inline Faust code in Pure programs. A complete example is shown in Section 6. Section 7 concludes with some remarks on the current status of the interface and possible

future enhancements.

2 The Faust backend

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 [5]. Install this on your system with the usual `make && sudo make install` commands.

The `-lang llvm` option instructs Faust to output LLVM bitcode (instead of the usual C++ code). Also, you want to add the `-double` option to make the compiled Faust module use double precision floating point values for samples and control values. So you'd compile an existing Faust module in the source file `example.dsp` as follows:

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

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 5).

Note that LLVM code actually comes in three distinct flavours:

- as an internal representation (*LLVM IR*), i.e., a C++ data structure in main memory used in most LLVM client applications such as compilers and interpreters;
- as a compact binary code (*LLVM bitcode*), which provides a serialized form of LLVM IR which can be passed from one LLVM application to another, either in main memory or as a disk file;
- and, last but not least, as a kind of human-readable assembler source code (*LLVM assembler*), which is rarely used directly in LLVM applications, but very useful for documentation purposes.

A description of the LLVM assembler code format can be found on the LLVM website [4], but the code examples shown in this paper should be rather self-explanatory, at least for C programmers. For the sake of a simple example, let us consider the following little Faust module which mixes two input signals and multiplies the resulting mono signal with a gain value supplied as a control parameter:

```
gain = nentry("gain", 0.3, 0, 10, 0.01);
process = + : *(gain);
```

From this the Faust compiler creates an LLVM bitcode file containing several LLVM assembler routines whose call interfaces are listed in Figure 1. If you want to see all the gory details, you can put the above code into a text file `example.dsp` and run Faust as follows to have it print the complete LLVM assembler code on standard output:

```
faust -double -lang llvm example.dsp
```

At the beginning of the LLVM module you see some data type definitions and global variables. The assembler routines roughly correspond to the various methods of the dsp classes Faust creates when generating C++ code. The central routine is `compute_llvm` which contains the actual assembler code for the signal processing function implemented by the Faust program. This routine gets invoked with the pointer to the dsp instance, the number of samples to be processed in one go (i.e., the block size), and the vectors of input and output buffers holding the sample values. The other routines are used for managing and inspecting dsp instances as well as the interface to the control variables (the "user interface" of a dsp in Faust parlance).

Note that the names of the assembler routines are currently hard-wired in Faust. Thus an LLVM application which wants to link in the Faust-generated code must be prepared to perform some kind of name mangling to make multiple Faust dsps coexist in a single LLVM module. This is handled transparently by Pure's bitcode loader.

3 The Pure bitcode interface

The nice thing about LLVM bitcode is that it can be readily loaded by LLVM applications and compiled to native machine code using the LLVM JIT compiler. This doesn't require any special linker utilities, only the LLVM library is needed.

The Pure compiler has a built-in bitcode loader which handles this. 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.

```
%struct.UIGlue = { ... }
%struct.dsp_llvm = type { double }

@fSamplingFreq = private global i32 0
@example = private constant [8 x i8] c"example\00"
@gain = private constant [5 x i8] c"gain\00"

define void @destroy_llvm(%struct.dsp_llvm* %dsp) { ... }
define void @delete_llvm(%struct.dsp_llvm* %dsp) { ... }
define %struct.dsp_llvm* @new_llvm() { ... }
define void @buildUserInterface_llvm(%struct.dsp_llvm* %dsp,
  %struct.UIGlue* %interface) { ... }
define i32 @getNumInputs_llvm(%struct.dsp_llvm*) { ... }
define i32 @getNumOutputs_llvm(%struct.dsp_llvm*) { ... }
define void @classInit_llvm(i32 %samplingFreq) { ... }
define void @instanceInit_llvm(%struct.dsp_llvm* %dsp,
  i32 %samplingFreq)  { ... }
define void @compute_llvm(%struct.dsp_llvm* %dsp, i32 %count,
  double** noalias %inputs, double** noalias %outputs) { ... }
define void @init_llvm(%struct.dsp_llvm* %dsp, i32 %samplingFreq) { ... }
```

Figure 1: Outline of the LLVM assembler code for a sample Faust module.

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

```
using "dsp:example";
```

The above statement loads the bitcode module, links it into the Pure program, and makes the Faust interface functions callable from Pure. It also mangles the function names and puts them into their own Pure namespace, so that different Faust modules can be called in the same Pure program. Note that it's not necessary to supply the `.bc` bitcode extension, it will be added automatically. Also, the bitcode module will be searched on Pure's library search path as usual. 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 look as shown in Figure 2. (You can also obtain this listing yourself by typing `show -g example::*` in the Pure interpreter after loading the module.) Note that despite the generic `struct_dsp_llvm` pointer type, the Pure compiler generates code that ensures that the dsp instances are fully typechecked at runtime. Thus it is only possible to pass a dsp struct pointer to the interface routines of the Faust module it was created with.

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 following section. 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 Running Faust dsps in Pure

Let's now have a look at how we can actually use a Faust module in Pure to process some samples. We present this in a cookbook fashion, using the `example.dsp` from the previous sections as a running example. We assume here that you already started the Pure interpreter in interactive mode (just run the `pure` command in the shell to do this), so the following input is meant to be typed at the '`>`' command prompt of the interpreter.

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

Figure 2: Call interfaces for the sample Faust module on the Pure side.

Step 1: Compile the Faust dsp We already discussed this in Section 2. You can execute the necessary command in the Pure interpreter using a shell escape as follows:

```
> ! faust -double -lang llvm example.dsp -o
example.bc
```

Step 2: Load the Faust dsp in Pure This was already covered in Section 3:

```
> using "dsp:example";
```

Please note that the first two steps can be omitted if you inline the Faust program in the Pure script, see Section 5.

Step 3: Create and initialize a dsp instance After importing the Faust module you can now create an instance of the Faust signal processor using the `newinit` routine, and assign it to a Pure variable as follows:

```
> let dsp = example::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. It's completely up to the dsp whether it actually uses this value in some way (our example doesn't, but we need to specify a value anyway).

The dsp is now fully initialized and we can use it to compute some samples. But before we can do this, we'll 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 = example::info dsp;
```

Step 4: Prepare input and output buffers Pure's Faust interface allows you to pass Pure double matrices as sample buffers, which makes this step quite convenient. For given numbers k and l of input and output channels, respectively, we'll need a $k \times n$ matrix for the input and a $l \times n$ matrix for the output, where n is the desired block size (the number of samples to be processed per channel in one go). Note that the matrices have one row per input or output channel. Here's how we can create some suitable input and output matrices using a Pure matrix comprehension and the `dmatrix` function available in Pure's standard library:

```
> let n = 10; // the block size
> let in  = {i*10.0+j | i = 1..k; j = 1..n};
> let out = dmatrix (l,n);
```

In our example, $k = 2$ and $l = 1$, thus we obtain the following matrices:

```
> in;
{11.0,12.0,13.0,14.0,15.0,16.0,17.0,18.0,19.0,20.0;
21.0,22.0,23.0,24.0,25.0,26.0,27.0,28.0,29.0,30.0}
> out;
{0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0}
```

Step 5: Apply the dsp to compute some samples With the `in` and `out` matrices as given above, we can now apply the dsp by invoking its `compute` routine:

```
> example::compute dsp n in out;
```

This takes the input samples specified in the `in` matrix and stores the resulting output in the `out` matrix. Let's take another look at the output matrix:

```
> out;
{9.6,10.2,10.8,11.4,12.0,12.6,13.2,13.8,14.4,15.0}
```

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` (possibly with different `in` buffers) to compute as much of the output signal as we need.

Step 6: Inspecting and modifying control variables Recall that our sample dsp also has a control variable `gain` which lets us change the amplification of the output signal. We've already assigned the corresponding information to the `ui` variable, let's have a look at it now:

```
> ui;
vgroup ("example",[nentry #<pointer 0xd81820>
("gain",0.3,0.0,10.0,0.01)])
```

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 containing a single `gain` parameter, which is represented as a Pure term containing the relevant information about the type, name, initial value, range and stepsize of the control, along with a `double` pointer which can be used to inspect and modify the control value. 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;
{"gain"=>#<pointer 0xd81820>}
```

The result is a Pure record value indexed by control names, thus the pointer which belongs to our `gain` control can be obtained with `ui!"gain"` (note that '`!`' is Pure's indexing operator). There are also convenience functions to inspect a control and change its value:

```
> let gain = ui!"gain";
> get_control gain;
0.3
> put_control gain 1.0;
()
> get_control gain;
1.0
```

Finally, let's rerun `compute` to get another block of samples from the same input data, using the new `gain` value:

```
> example::compute dsp n in out;
> out;
{32.0,34.0,36.0,38.0,40.0,42.0,44.0,46.0,48.0,
50.0}
```

As you can see, all these steps are rather straightforward. Of course, in a real program we would probably run `compute` in a loop which reads some samples from an audio device or sound file, applies the dsp, and writes back the resulting samples to another audio device or file. This can all be done quite easily in Pure using the appropriate addon modules available on the Pure website.

Also note that you could change the Faust source at any time, by editing the `example.dsp` file accordingly and returning to step 1. You don't even need to exit the Pure interpreter to do this.

5 Inlining Faust code

The process sketched out in the preceding section can be made even more convenient by inlining the Faust program in Pure. The Pure interpreter then handles the compilation of the Faust program automatically, invoking the Faust compiler when needed. (The command used to invoke the Faust compiler can be customized using the `PURE_FAUST` environment variable. The default is `faust -double`; the `-lang llvm` option is always added automatically.)

To add inline Faust code to a Pure program, the foreign source code is enclosed in Pure's inline code brackets, `%< ... %>`. You also need to add a '`dsp`' tag identifying the contents as Faust source, along with the name of the Faust module (which, as we've seen, becomes the namespace into which the Pure compiler places the Faust interface routines). The inline code section for our previous example would thus look as follows:

```
%< -*- dsp:example -*-
gain = nentry("gain", 0.3, 0, 10, 0.01);
process = + : *(gain);
%>
```

You can insert these lines into a Pure script, or just type them directly at the prompt of the Pure interpreter. If you later want to change the Faust source of the module, it is sufficient to just enter the inline code section again with the appropriate edits.

6 Example

As a more substantial but still self-contained example, Figures 3 and 4 show the source code of a complete stereo amplifier stage with bass, treble, gain and balance controls and a dB meter. The dsp part is implemented as inlined Faust code, as discussed in the previous section. The Pure part implements a Pd "tilde" object named

amp~. This requires 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. A sample patch showing this object in action can be seen in Figure 5.

A complete discussion of this example is beyond the scope of this paper, but note that the amp_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, this also includes setting the control parameters of the Faust dsp in response to incoming control messages, and the generation of output control messages to send the dB meter values (also computed in the Faust dsp) to Pd.

By using the interactive live editing facilities provided by pd-pure, we could now start adding more sophisticated control processing or even change the Faust program on the fly, while the Pd patch keeps running. We refer the reader to the pd-pure documentation for details [3].

7 Conclusion

The facilities described in this paper are fully implemented in the latest versions of the Pure and Faust compilers. We also mention in passing that Pure doesn't only support dynamic execution of mixed Pure and Faust code in its interactive interpreter environment, but Pure scripts containing Faust code can also be batch-compiled to native executables. This eliminates the JIT compilation phase and thus makes programs start up faster.

The present interface is still fairly low-level. Except for the automatic support for handling Faust control variables, the call interfaces to the Faust routines follows the code generated by Faust very closely. In the future, we might add more convenience functions at the Pure level which make the operation of Faust dsps easier for the Pure programmer.

Another 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 embed Faust as a domain-specific sublanguage in Pure. This would provide an alternative to other interactive signal processing environments based on Lisp dialects such as Snd-Rt [6].

Acknowledgements

Many thanks to Stéphane Letz 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.

References

[1] clang: a C language family frontend for LLVM. http://clang.llvm.org, 2011.

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

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

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

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

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

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

[8] D. A. Terei and M. M. Chakravarty. An LLVM backend for GHC. In *Proceedings of the third ACM SIGPLAN Haskell Symposium*, Haskell '10, pages 109–120, New York, NY, USA, 2010. ACM.

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

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

// bass and treble frequencies
bass_freq       = 300;
treble_freq     = 1200;
// bass and treble gain controls in dB
bass_gain       = nentry("bass", 0, -20, 20, 0.1);
treble_gain     = nentry("treble", 0, -20, 20, 0.1);
// gain and balance controls
gain            = db2linear(nentry("gain", 0, -96, 96, 0.1));
bal             = hslider("balance", 0, -1, 1, 0.001);
// stereo balance
balance         = *(1-max(0,bal)), *(1-max(0,0-bal));

// generic biquad filter
filter(b0,b1,b2,a0,a1,a2)       = f : (+ ~ g)
with {  f(x)    = (b0/a0)*x+(b1/a0)*x'+(b2/a0)*x'';
        g(y)    = 0-(a1/a0)*y-(a2/a0)*y'; };

/* Low and high shelf filters, straight from Robert Bristow-Johnson's "Audio
   EQ Cookbook". */

low_shelf(f0,g)         = filter(b0,b1,b2,a0,a1,a2)
with {  S  = 1; A = pow(10,g/40); w0 = 2*PI*f0/SR;
        alpha = sin(w0)/2 * sqrt( (A + 1/A)*(1/S - 1) + 2 );
        b0 =    A*( (A+1) - (A-1)*cos(w0) + 2*sqrt(A)*alpha );
        b1 =  2*A*( (A-1) - (A+1)*cos(w0)                   );
        b2 =    A*( (A+1) - (A-1)*cos(w0) - 2*sqrt(A)*alpha );
        a0 =        (A+1) + (A-1)*cos(w0) + 2*sqrt(A)*alpha;
        a1 =   -2*( (A-1) + (A+1)*cos(w0)                   );
        a2 =        (A+1) + (A-1)*cos(w0) - 2*sqrt(A)*alpha; };
high_shelf(f0,g)        = filter(b0,b1,b2,a0,a1,a2)
with {  S  = 1; A = pow(10,g/40); w0 = 2*PI*f0/SR;
        alpha = sin(w0)/2 * sqrt( (A + 1/A)*(1/S - 1) + 2 );
        b0 =    A*( (A+1) + (A-1)*cos(w0) + 2*sqrt(A)*alpha );
        b1 = -2*A*( (A-1) + (A+1)*cos(w0)                   );
        b2 =    A*( (A+1) + (A-1)*cos(w0) - 2*sqrt(A)*alpha );
        a0 =        (A+1) - (A-1)*cos(w0) + 2*sqrt(A)*alpha;
        a1 =    2*( (A-1) - (A+1)*cos(w0)                   );
        a2 =        (A+1) - (A-1)*cos(w0) - 2*sqrt(A)*alpha; };

// the tone control
tone            = low_shelf(bass_freq,bass_gain)
                : high_shelf(treble_freq,treble_gain);

// envelop follower (1 pole LP with configurable attack/release time)
t               = 0.1;                  // attack/release time in seconds
g               = exp(-1/(SR*t));       // corresponding gain factor
env             = abs : *(1-g) : + ~ *(g) : linear2db;

// dB meters for left and right channel (passive controls)
left_meter(x)   = attach(x, env(x) : hbargraph("left", -96, 10));
right_meter(x)  = attach(x, env(x) : hbargraph("right", -96, 10));

// the main program of the Faust dsp
process         = (tone, tone) : (_*gain, _*gain) : balance
                : (left_meter, right_meter);
%>
```

Figure 3: Amplifier plugin, Faust part.

```
// These are provided by the Pd runtime.
extern float sys_getsr(), int sys_getblksize();
// Provide some reasonable default values in case the above are missing.
sys_getsr = 48000; sys_getblksize = 64;

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

using faustui, system;

amp_dsp = k,l,amp with
  // The dsp loop. This also outputs the left and right dbmeter values for
  // each processed block of samples on the control outlet, using messages of
  // the form left <value> and right <value>, respectively.
  amp in::matrix = amp::compute dsp n in out $$
    out,[left (get_control left_meter),right (get_control right_meter)];
  // Respond to control messages of the form <control> <value>. <control> may
  // be any of the input controls supported by the Faust program (bass,
  // treble, gain, etc.).
  amp (c@_ x::double) = put_control (ui!str c) x $$ x;
end when
  // Initialize the dsp.
  dsp = amp::newinit SR;
  // Get the number of inputs and outputs and the control variables.
  k,l,ui = amp::info dsp;
  ui = control_map $ controls ui;
  {left_meter,right_meter} = ui!!["left","right"];
  // Create a buffer large enough to hold the output from the dsp.
  out = dmatrix (l,n);
end;
```

Figure 4: Amplifier plugin, Pure part.

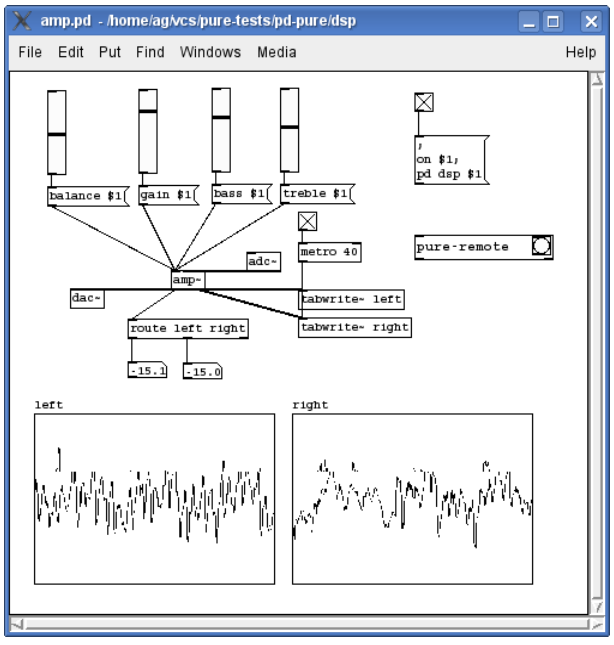

Figure 5: Amplifier plugin, Pd patch.