# Term Rewriting Extension for the Faust Programming Language

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

## Abstract

This paper discusses a term rewriting extension for the functional signal processing language Faust. The extension equips Faust with a hygienic macro processing facility. Faust macros can be used to define complicated, parameterized block diagrams, and perform arbitrary symbolic manipulations of block diagrams. Thus they make it easier to create elaborate signal processor specifications involving many complicated components.

## Keywords

Digital signal processing, Faust, functional programming, macro processing, term rewriting.

## 1 Introduction

Faust is a functional signal processing language, which is used to develop digital signal processors handling synchronous streams of sample values. It is mostly targeted at audio and music applications at this time. Faust has a formal semantics (based on the lambda calculus and a block diagram algebra) which means that it can be used as a specification language for describing signal proccessors in an implementation-independent way. These specifications are executable, however, and Faust provides sophisticated optimizations and compilation to C++ to turn the specifications into efficient code which can compete with carefully hand-crafted routines. Faust works with an abundance of different platforms and plugin environments such as Jack, LADSPA, VST, Pd, Max and several programming languages, just a recompile is enough to create code for the various architectures. Last but not least, the Faust compiler also provides automatic documentation facilities which produce block diagrams in SVG format and LaTeX documents.

Faust has been discussed at the Linux audio conference and elsewhere on various occasions [3; 1] and is quickly gaining traction in the signal processing community. A description of the formal underpinnings can be found in [2]. Faust is free software distributed under the GPL V2+, see `http://faust.grame.fr`.

This paper reports on the Faust term rewriting extension which equips the language with a kind of hygienic macro facility for specifying complex block diagrams in a more convenient fashion. Macros are defined by rewriting rules and are expanded away at compile time by rewriting terms in the Faust block diagram algebra. The resulting Faust program is then compiled as usual. This facility has been developed by the author in collaboration with Yann Orlarey, the principal author of Faust. It has already been available in recent Faust versions for quite some time but has never been documented anywhere; this paper attempts to fill this gap.

## 2 Basic Faust Example

Faust works on *sampled signals* which are thought of as functions mapping (discrete) time to sample values. Signals can be constant or time-varying, or they can be *control signals* embodied by special elements which are typically implemented as GUI, MIDI and/or OSC controls, depending on the target architecture. The following listing shows a simple Faust program which implements a sine tone generator:

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

vol  = nentry("vol", 0.3, 0, 10, 0.01);
pan  = nentry("pan", 0.5, 0, 1, 0.01);
freq = nentry("pitch", 440, 20, 20000, 0.01);

process = osc(freq)*vol : panner(pan);
```

The example features three control elements for specifying volume, panning and frequency. The sine signal created by the `osc` function gets multiplied by the volume and then passed through a panner which turns it into a stereo output signal. The `process` function is the "main" function of a Faust program; it denotes the signal processing function realized by the program. In this case, the `process` function has

no arguments and thus the implemented signal processor has no inputs. In general, any Faust function (including `process`) can have an arbitrary number of input and output signals.

## 3 The Term Rewriting Extension

Faust signal processors are essentially *terms* in the Faust *block diagram algebra* (BDA) which is described elsewhere [2]. Briefly, the BDA consists of algebraic operations (written as infix and postfix operators) which specify various combinations of signal processing functions, in particular:

- `f'` and `f@n` *delay* `f` by one and a given number `n` of samples, respectively.
- `f:g` and `f,g` specify the *serial* and *parallel* composition of two signal processing functions.
- `f<:g` and `f:>g` *split* and *merge* (mix) the outputs of `f` and route them to corresponding inputs of `g`.
- `f~g` combines `f` and `g` to a *loop* with implicit 1-sample delay.

In addition, the usual arithmetic, logical and comparison operators (`+`, `*`, etc.) as well as mathematical functions (`exp`, `sin`, etc.) work on signals in a pointwise fashion. Thus, e.g., `f+g` is the signal obtained by adding each sample of the signals `f` and `g`.

Term rewriting provides us with a means to manipulate these BDA terms in an algebraic fashion at compile time. For instance, the following two rewriting rules define a macro named `fact` which implements the factorial:

```
fact(0) = 1;
fact(n) = n*fact(n-1);
process = fact(3);
```

The last line in this program gives the usual `process` function of a Faust program. The resulting signal processor outputs the constant signal $3! = 6$.

Note that Faust doesn't have a special keyword for denoting macro definitions, instead these are flagged by employing *patterns* on the left-hand side of such a rule, which can be constants (such as the constant `0` in this example), variables (such as `n`) and arbitrary expressions formed with these and the BDA operations. That is, a pattern is an arbitarily complex BDA expression involving variables and constants. At least one of the macro arguments must be a nontrivial pattern (i.e., not simply a variable), otherwise the definition will be taken to be an ordinary function definition. Also note that macro definitions may involve multiple rewriting rules, as in the example above.

Just like Faust's functions are nothing but named lambdas, there are also *anonymous macros* which take the form of a **`case`** expression listing all the argument patterns and the corresponding substitutions. For instance, the above definition of the factorial macro is actually equivalent to:

```
fact = case { (0) => 1; (n) => n*fact(n-1); };
```

## 4 Macro Evaluation by Rewriting

Rewriting rules are applied by matching them to BDA terms on the right-hand side of function definitions. To these ends, the rules are considered in the order in which they are written in the Faust program, binding variables in the left-hand side of rules to the corresponding values. Evaluation proceeds from left to right, innermost expressions first. Thus, in the above example the term `fact(3)` in the definition of the `process` function will be rewritten to the constant `6`, using the following reduction sequence:

```
fact(3) → 3*fact(3-1) → 3*fact(2)
→ 3*(2*fact(2-1)) → 3*(2*fact(1))
→ 3*(2*1*fact(1-1))) → 3*(2*(1*fact(0)))
→ 3*(2*(1*1))) → 6
```

Note that in this process the Faust compiler also evaluates constant signal expressions such as `3-1` → `2` and `3*(2*(1*1)))` → `6`.

Another example, which employs pattern matching on BDA operations, is the following little macro `serial` which turns parallel compositions into serial ones:

```
serial((x,y)) = serial(x) : serial(y);
serial(x)     = x;
process       = serial((sin,cos,tan));
```

The result is the same as if you had written the serial composition `sin:cos:tan`, cf. Fig. 1.

As the above examples show, macro definitions can also be recursive, making it possible to analyze and build arbitrarily complicated BDA terms. Here is another example, which employs a variation of the `fold` operation (customary in functional programming libraries) to accumulate values. In this case the values are actually signals, produced by a function (or macro) `x` which maps a running index to a signal. This

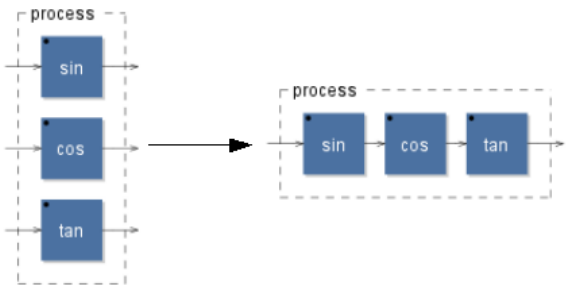


Figure 1: Parallel-serial conversion macro.

allows us to emulate Faust's `sum` function which adds up an arbitrary collection of signals:

```
fold(1,f,x) = x(0);
fold(n,f,x) = f(fold(n-1,f,x),x(n-1));
fsum(n)     = fold(n,+);

f0 = 440; a(0) = 1; a(1) = 0.5; a(2) = 0.3;
h(i)    = a(i)*osc((i+1)*f0);
v       = hslider("vol", 0.3, 0, 1, 0.01);
process = v*fsum(3,h);
```

The resulting signal processor (a simple additive synthesizer which adds up three harmonics), is shown in Fig. 2. Note that the three oscillators and their amplitudes are also defined using rewriting rules in this example.

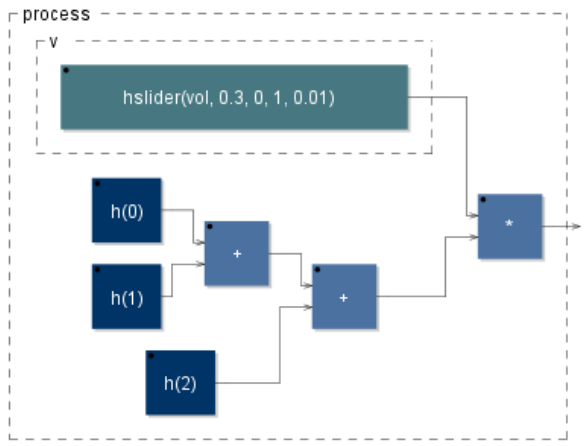


Figure 2: Sum macro example.

## 5 A Note on Macro Hygiene

It is worth noting the differences between Faust's rewriting-based macros and the textual macros in languages such as C. In the latter case, macros are just textual substitutions which is very flexible but also has various shortcomings, most notably *name capture*. For instance, take the following C macro definition:

```
#define F(x) { int y = x+1; return x+y; }
```

Given this definition, `F(y)` expands to `{ int y = y+1; return y+y; }` which is usually *not* what you want.

Faust macros do not have any such pitfalls since they work on the internal representation of BDA terms rather than the program text. This means that variables on the left-hand side of a macro rule are always bound *lexically*, i.e., according to the block scoping rules defined by the Faust language. Thus a Faust macro like `F = case {(x) => x+y with { y = x+1; };}` (which is roughly equivalent to the C macro above, minus the name capture) will work correctly no matter what gets passed for the macro parameter `x` (even if it is a signal named `y`). Macros with this desirable property are also called *hygienic macros* in the programming language literature.

## 6 Example: A Systolic Array

The following program illustrates how to use macros in order to abbreviate complicated, parameterized block diagrams. The example we consider is a kind of "systolic array", a grid of binary operations organized in a mesh (Fig. 3).

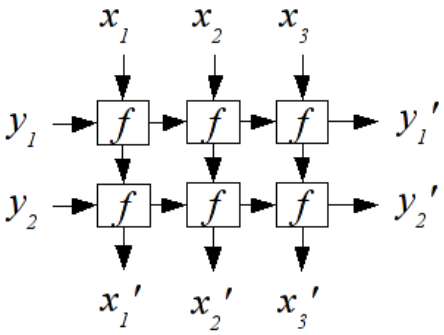


Figure 3: Systolic array.

The Faust program used to produce this layout is given below.

```
g(1,f)  = f;
g(m,f)  = (f, r(m-1)) : (_, g(m-1,f));

h(1,m,f) = g(m,f);
h(n,m,f) = (r(n+m) <: (!,r(n-1),s(m),
            (_,s(n-1),r(m) : g(m,f)))) :
           (h(n-1,m,f), _);

r(1) = _; r(n) = _,r(n-1); // route through
s(1) = !; s(n) = !,s(n-1); // skip

f = + <: _,_; // sample cell function
process = h(2,3,f);
```

Note that the macro `g` constructs a single row of the mesh for the given number $m$ of grid cells and the given function `f` which takes two input signals and produces two output signals. The macro `h` applies `g` repeatedly in order to build an $n \times m$ mesh from its rows. Two helper macros `r` and `s` perform the necessary routing between the components. These employ two basic elements of the BDA, '`_`' which simply routes through a single input to a single output (i.e., `_(x) = x`), and '`!`' which denotes a "sink" for a single input (`!(x) = ()`). (A more detailed explanation of the construction is given below.)

The given `process` function illustrates the use of the `h` macro to construct a $2 \times 3$ "accumulator" mesh from the cell function `f = + <: _,_` which just adds its two inputs and sends the computed sum to its two outputs. A Pd patch showing this signal processor in action is shown in Fig. 4.

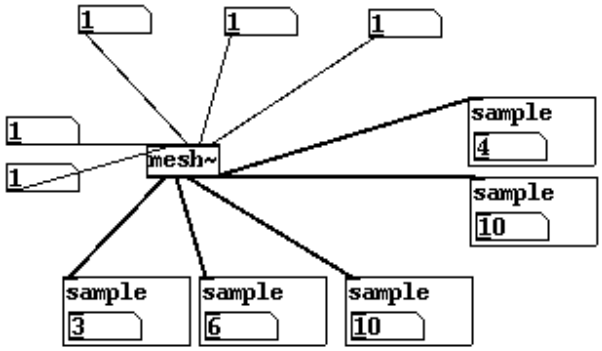


Figure 4: Systolic array example in a Pd patch.

To see how the recursive construction works, note that there are two equations for the `h` macro. The first equation deals with the case of one row. This is handled by just invoking the macro `g` which in turn applies the function `f` once (digesting the single row input $y_1$ and the first column input $x_1$) and invokes itself recursively on the first output of `f` and the remaining inputs $x_2, \ldots, x_m$ (which are routed through with the `r` macro). The base case $m = 1$ is treated in the first equation for `g` which just yields `f` itself.

The interesting case is the second equation for `h` in which we deal with more than one row, cf. Fig. 5. Here the input signal, consisting of $n$ row inputs $y_1, \ldots, y_n$ and $m$ column inputs $x_1, \ldots, x_m$, gets split up in two parts. The expression `!,r(n-1),s(m)` gives $y_2, \ldots, y_n$ which is simply routed through. The expression `_,s(n-1),r(m)` yields $y_1, x_1, \ldots, x_m$ which is piped into `g`, producing a single row of the mesh. The signals $y_2, \ldots, y_n$ are then recombined with the first $m$ outputs $x'_1, \ldots, x'_m$ of `g`, and the result is passed to `h` to recursively construct the remaining mesh of $n-1$ rows, yielding the output signals $x''_1, \ldots, x''_m; y'_n, \ldots, y'_2$. Tacking on the remaining output signal $y'_1$ of the call to `g` gives us the final result $x''_1, \ldots, x''_m; y'_n, \ldots, y'_1$.

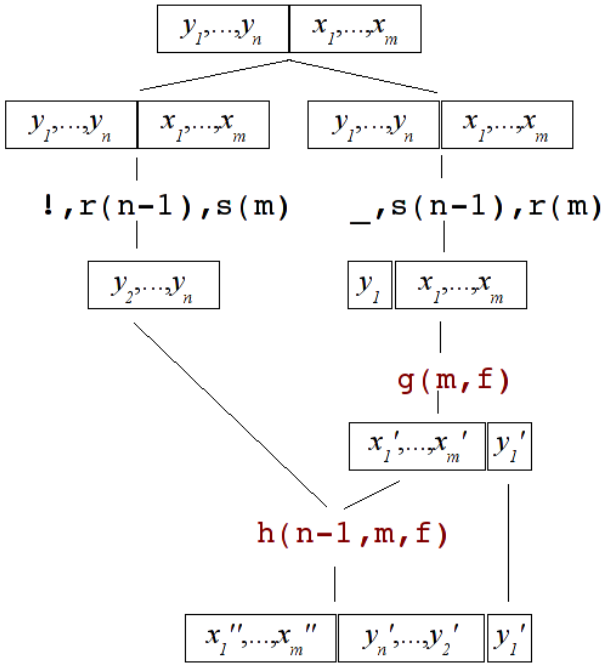


Figure 5: Recursive construction in the systolic array macro.

There are still some ways in which the `h` macro can be improved. For programming convenience and simplicity, `h` takes its inputs as $y_1, \ldots, y_n; x_1, \ldots, x_m$, with the row inputs coming first, and produces the row outputs $y'_n, \ldots, y'_1$ in reverse order. In fact, this layout of arguments and results is quite convenient in the Pd patch. However, as a programmer using this macro in your Faust programs you'd probably prefer a macro which maps an $m+n$ tuple of input signals $x, y$ to a corresponding tuple $x', y'$ of output signals in the "right" order. Fortunately, adding this functionality as separate pre- and postprocessing stages by making good use of the `r` and `s` macros is fairly easy. We leave this as an exercise to the interested reader.

## 7 Conclusion

The term rewriting extension sketched out in this paper equips Faust with a simple macro processing facility which is useful to define abbreviations for complicated, parameterized block diagrams, and to perform arbitrary symbolic ma-

nipulations on block diagrams in the preprocessing stage of the Faust compiler. To these ends, terms in the Faust block diagram algebra (BDA) are rewritten using term rewriting rules. Evaluating macro invocations using the provided rules is performed by Faust at compilation time. Faust's term rewriting macros are *structured* (they operate on term structures rather than program text) and *hygienic*, i.e., all bindings of macro variables are performed lexically, and thus Faust macros are not susceptible to "name capture" which make less sophisticated macro facilities in languages such as C bug-ridden and hard to use.

There are still some shortcomings in Faust's macro system which will hopefully be addressed in the future:

- Faust does its own normalizations of BDA terms "under the hood" and thus it can be hard to figure out exactly which patterns are needed to rewrite certain constructs of the Faust language.
- Only plain term rewriting rules are supported at this time. Adding conditional (i.e., guarded) rules would make the system more versatile.
- It would be useful to provide an interface to Faust's block diagram optimization pass so that custom optimization rules could be implemented using macros.

## References


[1] A. Gräf. Interfacing Pure Data with Faust. In *5th International Linux Audio Conference*, pages 24–31, Berlin, 2007. TU Berlin.

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

[3] Y. Orlarey, A. Gräf, and S. Kersten. DSP programming with Faust, Q and SuperCollider. In *4th International Linux Audio Conference*, pages 39–47, Karlsruhe, 2006. ZKM.