# Functional Framework for Sound Synthesis

**Conference Paper** *in* Lecture Notes in Computer Science · January 2005

DOI: 10.1007/978-3-540-30557-6_3 · Source: DBLP



**1 author:**

Jerzy Karczmarczuk
Université de Caen Normandie
**43** PUBLICATIONS **238** CITATIONS

SEE PROFILE

# Functional Framework for Sound Synthesis

Jerzy Karczmarczuk

Dept. of Computer Science, University of Caen, France
karczma@info.unicaen.fr

**Abstract.** We present an application of functional programming in the domain of sound generation and processing. We use the lazy language Clean to define purely functional stream generators, filters and other processors, such as reverberators. Audio signals are represented (before the final output to arrays processed by the system primitives) as co-recursive lazy streams, and the processing algorithms have a strong dataflow taste. This formalism seems particularly appropriate to implement the 'waveguide', or 'physically-oriented' sound models. Lazy programming allocates the dynamical memory quite heavily, so we do not propose a real-time, industrial strength package, but rather a pedagogical library, offering natural, easy to understand coding tools. We believe that, thanks to their simplicity and clearness, such functional tools can be also taught to students interested in audio processing, but with a limited competence in programming.

**Keywords:** Lazy streams, Sounds, DSP, Clean.

## 1 Introduction

The amplitude of a sound (for one channel) may be thought of as a real function $f$ of time $t$, and it is fascinating how much structural information it may contain [1]. In order to produce some audible output, this function must be sampled, and transformed into a *signal*, and this is the basic data type we shall work on. Sound may be represented at many different levels, and if one is interested in the structure of sequences of musical events, chords, phrases, etc., there is no need to get down to the digital signal processing primitives. It may seem more interesting and fruitful to speak about the algebra of musical events, music combinators, etc. This was the idea of Haskore [2], whose authors used Haskell to define and to construct a whole spectrum of musical "implementable abstractions". Haskore deals with high-level musical structures, and consigns the low-level, such as the interpretation of the MIDI streams, or the spectral structure of sounds to some back-end applications, MIDI players or CSound [3].

We decided to use the functional approach for the specification and the coding of this "low end" sound generation process. This is usually considered a highly numerical domain involving filter and wave-guide design [4], Fourier analysis, some phenomenological "magic" of the Frequency Modulation approach [5], or some models based on simplified physics, such as the Karplus-Strong algorithm [6] for the plucked string, and its extensions. But the generation and transformation of sounds is a *constructive* domain, dealing with complex abstractions (such as timbre, reverberation, etc.), and it may be based on a specific algebra as well. A possible application of functional programming paradigms as representation and implementation tools seems quite natural.

Software packages for music and acoustics are *very* abundant, some of them are based on Lisp, such as CLM [7] or Nyquist [8], and their authors encourage the functional style of programming, although the functional nature of Lisp doesn't seem very important therein as the implementation paradigm. But the the interest of the usage of functional tools in the DSP/audio domain is definitely growing, see the article [9], and the HaskellDSP package of Matt Donadio [10].

We don't offer yet a fully integrated application, but rather a pedagogical library called *Clarion*[1], in the spirit of, e.g., the C++ package STK de Perry Cook [11], but — for the moment — much less ambitious, demonstrating mainly some models based on the waveguide approach [4]. It has been implemented in a pure, lazy functional language Clean [12], very similar to Haskell. Its construction focuses on:

- Co-recursive (lazy) coding of "data flow" algorithms with delay lines, filters, etc. presented declaratively. This is the way we build the sound samples.
- Usage of higher-order combinators. We can define "abstract" instruments and other sound processors at a higher level than what we see e.g., in Csound.

The dataflow approach to music synthesis, the construction of virtual synthesizers as collections of pluggable modules is of course known, such systems as Buzz, PD or Max/JMax [13–15] are quite popular. There is also a new sequential/time-parallel language Chuck [16] with operators permitting to transfer the sound data streams between modules. All this has been used as source of inspiration and of algorithms, sometimes easier to find in library sources than in publications...

Our implementation does not neglect the speed issues, but because of dynamic memory consumption the package is not suitable for heavy duty event processing. It is, however, an autonomous package, able to generate audible output on the Windows platform, to store the samples as `.wav` files, and to read them back. It permits to code simple music using a syntax similar to that of Haskore, and to depict graphically signals, envelopes, impulse responses, or Fourier transforms. The choice of Clean has been dictated by its good interfacing to lower-level Windows APIs, without the necessity of importing external libraries. The manipulation of 'unique' unboxed byte arrays which ultimately store the sound samples generated and processed as streams, seems sufficiently efficient for our purposes. The main algorithms are platform-independent, and in principle they could be coded in Haskell. The main purpose of this work is pedagogical, generators and transformers coded functionally are compact and readable, and their cooperation is natural and transparent.

### 1.1 Structure of this article

This text can be assimilated by readers knowing typical functional languages such as Clean or Haskell, but possessing also some knowledge of the digital signal processing. While the mastery of filters, and the intricacies of such effects as the reverberation are not necessary in order to grasp the essential, some acquaintance with the signal processing techniques might be helpful.

[1] We acknowledge the existence of other products with this name.

We begin with some typical co-recursive constructions of infinite streams, and we show how a dataflow diagram representing an iterative signal processing is coded as a co-recursive, lazy stream. We show the construction of some typical filters, and we present some instruments, the plucked string implemented through the Karplus-Strong model, and simplified bowed string and wind instruments. We show how to distort the "flow of time" of a signal, and we show a simple-minded reverberator.

## 2 Co-recursive stream generators

A *lazy* list may represent a *process* which is activated when the user demands the next item, and then suspends again. This allows to construct infinite *co-recursive* sequences, such as $[0, 1, 2, 3, \ldots]$, coded as **`ints = intsFrom 0 where intsFrom n=[n:intsfrom (n+1)]`**, and predefined as **`[0 .. ]`**. Such constructions are useful e.g., for processing of infinite power series [18], or other iterative entities omnipresent in numerical calculi. The "run-away" co-recursion is not restricted to functions, we may construct also co-recursive ***data***. The sequence above can be obtained also as

```
ints = [0 : ints+ones] where ones=[1 : ones]
```

provided we overload the **`(+)`** operator so that it add lists element-wise. Here the sequences $y_n$ for $n = 0, 1, 2, \ldots$ will represent samples of a discretized acoustic signal, and for the ease of processing they will be real, although during the last stage they are transformed into arrays of 8- or 16-bit integers. Already with **`[0 .. ]`** we can construct several sounds, e.g., a not very fascinating sinusoidal wave:

```
wave = map (\n -> sin(2*Pi*n*h)) [0 .. ]
```

where **`h`** is the sampling period, the frequency divided by the sampling rate: the number of samples per second. In order to cover the audible spectrum this sampling rate should be at least 32000, typically it is equal to 44100 or higher.

In the next section we shall show how to generate such a monochromatic wave by a recursive formula without iteration of trigonometric functions, but the essential point is not the actual recipe. What is important is the fact the we have a *piece of data*, that an infinitely long signal is treated declaratively, outside any loop or other syntactic structure localized within the program. For efficiency we shall use a Clean specific type: head-strict unboxed lists of reals, denoted by such syntax: **`[# 1.0,0.8, ...]`**.

## 3 Simple periodic oscillator

One of the best known algorithmic generators of sine wave is based on the recurrent formula: $\sin(n\omega h) = 2\cos(\omega h)\sin((n-1)\omega h) - \sin((n-2)\omega h)$, where $h$ is some sampling interval, and $\omega = 2\pi f$ with $f$ being the frequency. One writes the difference equation obeyed by the discrete sequence: $y_n = c \cdot y_{n-1} - y_{n-2}$ where $c = 2\cos(\omega h)$ is a constant. One may present this as a dataflow "circuit" shown on Fig. (1). Here $z^{-1}$ denotes conventionally the unit delay box.

Of course, the circuit or the sequence should be appropriately initialized with some non-zero values for $y_0$ and $y_1$, otherwise the generator would remain silent. A Clean infinite list generator which implements the sequence $\sin(2\pi ft)$ with $t = n/\mathrm{SamplingRate}$ may be given in a typically functional, declarative style as

```
oscil fr = y where  //  DpiSR is 2π/SR
  omh = DpiSR*fr    //  (divisor: global SamplingRate)
  y = [# 0.0 : v]
  v = [# sin omh : ((2.0*cos omh)*>v - y)]
```

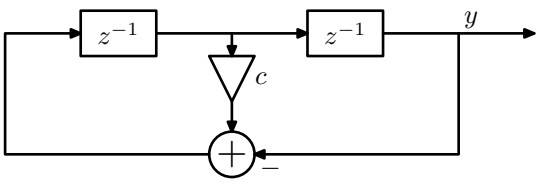


**Fig. 1.** Sine wave generator

The values $0.0 = \sin(0.0)$ and $\sin(\omega h)$ can be replaced by others if we wish to start with another initial phase. Note the cycle in the definition of variables, without laziness it wouldn't work! The operator **(*>)** multiplies a scalar by a stream, using the standard **Map** functional (overloading of **map** for **[#]** lists): **(*>) x l = Map (\s->x*s) l**; we have also defined a similar operator **(+>)**, which raises the values in a stream by a constant, etc. The subtraction and other binary arithmetic operators have been overloaded element-wise for unboxed real streams, using the known functional **zipWith**. Some known list functionals such as **map**, **iterate** or **take** in this domain are called **Map**, **Iterate**, etc. This algorithm is numerically stable, but in general, in presence of feedback one has to be careful, and *know* the behaviour of iterative algorithms, unless one wants to exploit the imprevisible, chaotic generators. . . . Another stable co-recursive algorithm may be based on a *modified* Euler method of solving the oscillator differential equation: $\ddot{y} = -\omega^2 y$ (through $\dot{y} = \omega v;\ \dot{v} = -\omega y$). We get $y$ equal to the sampled sine, with **y=[#0.0:y]+a*>v**; **v=[#1.0:v-a*>y]**. A pure sinusoidal wave is not a particularly fascinating example, but it shows already some features of our approach.

- As already mentioned, the sound signal is not a 'piece of algorithm' such as a loop over an array, but a first-class data item, although its definition contains a never-ending, recurrent code. We can add them in the process of additive synthesis, multiply by some scalars (amplify), replicate, etc. The modularity of the program is enhanced with respect to the imperative approach with loops and re-assignments. Some more complicated algorithms, such as the pitch shift are also easier to define.
- We have a *natural* way of defining data which rely on feedback. The coding of IIR filters, etc., becomes easy. In general, lazy streams seem to be a good, *readable* choice for implementing many 'waveguide' models for sound processing [4].

With the stream representation it is relatively easy to modulate the signal by an envelope, represented as another stream, typically of finite length, and normalized so that its maximum is equal to 1, it suffices to multiply them element-wise. An exponential decay (infinite) envelope can be generated on the spot as **expatt r = Iterate ((*) r) 1.0**. The *parameterization* of an arbitrary generator by an arbitrary envelope is more involved, and depends on the generator. But, often when in definitions written in this

style we see **`c*>s`** where **`c`** is a constant, we may substitute **`m*s`** for it, where **`m`** is the modulating stream. Thus, in order to introduce a 5 Hz *vibrato* (see fig. 9) to our oscillator, it suffices to multiply **`v`** in **`{2*cos omh*>v}`** by, say, **`{(1.0+>0.001*>oscil 5.0))}`**, since the concerned gain factor determines the pitch, rather than the amplitude. This is not a universal way to produce a varying frequence, we just wanted to show the modularity of the framework. Some ways of generating *vibrato* are shown later.

It is worth noting that treating signals as data manipulated arithmetically permits to code in a very compact way the instruments defined through the additive synthesis, where the sound is a sum of many partials (amplitudes corresponding to multiples of the fundamental frequency), for example:

```
additive freq amplist = y
 where
  m = [1.0 .. toReal(length amplist)]
  y = Foldl (+) zeros        // an infinite stream of zeros
      (map (\(n,a)->a*>oscil(n*freq)) (zip2 m amplist))

oboe fr= additive fr [0.0021,0.0237,0.1,0.0513,0.045,0.061,0.0168]
tuba fr= additive fr [0.10095,0.15732,0.14992,0.09895,0.07178,...]
```

Envelopes, *vibrato*, etc. effects can be added during the post-processing phase.

## 4 Some typical filters and other simple signal processors

A filter is a *stream processor*, a function from signals to signals, $x \rightarrow y$. The most simple smoothing, low-pass filter is the averaging one: $y_n = 1/2(x_n + x_{n-1})$. This can be implemented directly, in an index-less, declarative style, as **`y = 0.5*>(x + Dl x)`**, where **`Dl x = [#0.0 : x]`**. If we are not interested in the first element, this is equivalent to **`y = 0.5*>(x + Tail x)`**, but this variant is less lazy. This is an example of a FIR (Finite Impulse Response), non-recursive filter. We shall also use the "Infinite Response", recursive, feedback filters (IIR), where $y_n$ depends on $y_{n-k}$ with $k > 0$, and whose stream implementation is particularly compact.

One of the standard high-pass filters, useful in DSP, the "DC killer", 1-zero, 1-pole filter, which removes the amplitude bias introduced e.g., by integration, is described by the difference equation $y_n = b \cdot (x_n - x_{n-1}) + a \cdot y_{n-1}$, and constitutes thus a (slightly renormalized) differentiator, followed by a "leaky" integrator, everything depicted in Fig. 2. Here $a$ is close to 1, say $0.99$, and $b = (1 + a)/2$.

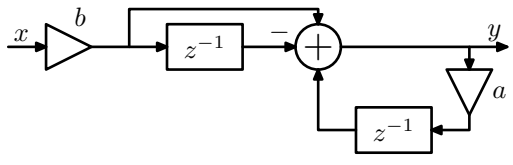


**Fig. 2.** DC blocker circuit

The effect of the filter on, say, a sample of brown (integrated white) noise is depicted on Fig. (3), with $b = 0.97$, and the equivalent Clean program is shown below, again, it is a co-recursive 1-liner, where a stream is defined through itself, delayed. This delay is essential in order to make the algorithm effective; our code, as usually in DSP, should not contain any "short loops", where a recursive definition of a stream cannot resolve the value of its head.

```
dcremove a (xs=:[# x0:xq]) = y where
  y=a*>Dl y+((1.0+a)/2.0)*>(xq-xs)
```

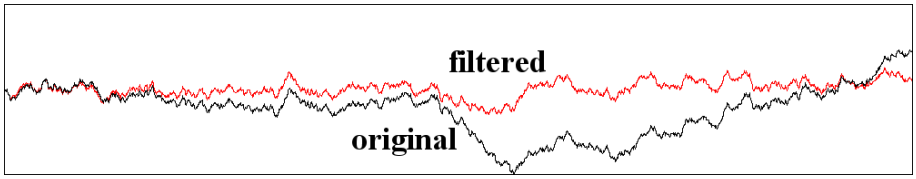


**Fig. 3.** DC removal from the "brown noise"

The damping sensitivity, and losses are bigger when $a$ is smaller. Note the difference between the diagram and the code: The delay and the multiplication by $b$ have been *commuted*, which is often possible for time-independent, linear modules, and here economises one operation. The parameter pattern syntax **a=:**$\alpha$ is equivalent to Haskell **a@**$\alpha$, and permits to name and to destructure a parameter at the same time.

Another feedback transducer of general utility is an 'all-pass filter', used in reverberation modelling, and always where we need some dispersion — difference of speed between various frequency modes. The variant presented here is a combination of 'comb' forward and feedback filters, and it is useful e.g. for the transformation of streams generated by strings (piano simulators, etc.). The definition of a simple all-pass filter may be

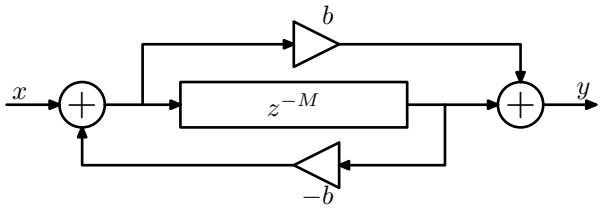


**Fig. 4.** An all-pass filter

```
allpass m b x = b*>z + v where
  v = delay m z   // Prepend m zeros to z
  z = x - b*>v
```

## 5 The Karplus-Strong model of a plucked string

Finally, here is another sound generator, based on a simplified 'waveguide' physical model [4]. A simplistic plucked string (guitar- or harpsichord-like sound) is constructed as a low-pass-filtered delay line with feedback. Initially the line is filled-up with any values, the (approximate) "white noise": a sequence of uncorrelated random values between $\pm 1$, is a possible choice, standard in the original Karplus-Strong model.

On output the neighbouring values are averaged, which smooths down the signal, and the result is pumped back into the delay line. After some periods, only audible frequencies are the ones determined by the length of the delay line, plus some

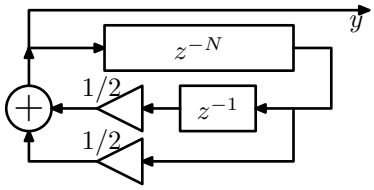


**Fig. 5.** Plucked string

harmonics. Higher frequencies decay faster than the lower ones, which is the typical feature of the plucked string, so this incredibly simple model gives a quite realistic sound! The implementation is straightforward, the only new element is the usage of a finite-length segment and the overloaded concatenation operator **(++|)**.

```
karstr f = y where
  prfx = Take (toInt (SR/f)) whitenoise
  y = prfx ++| 0.5 *> (y + Tl y)  // or more delayed: y+Dl y
```

The noise signal is a stream generated by an iterated random number generator. The package disposes of several noise variants, brown, pink, etc., useful for the implementation of several instruments and effects.

A more complicated string (but equally easy to code from a given diagram), with external excitation, some slightly detuned parallel delay lines, additional all-pass filters etc., is used to simulate piano, harpsichord or mandolin, but also bowed instruments (violin), and some wind instruments, since mathematically, strings and air columns are similar. We shall omit the presentation of really complex instruments, since their principle remains the same, only the parameterization, filtering, etc. is more complex.

Since a looped delay line will be the basic ingredient of many instruments, it may be useful to know where the model comes from. If $y(x,t)$ denotes the displacement of air or metal, etc. as a function of position (1 dimension) and time, and if this displacement fulfils the wave equation without losses nor dispersion (stiffness): $\partial^2 y/\partial t^2 = c^2 \cdot \partial^2 y/\partial x^2$, its general solution is a superposition of two *any* travelling waves: $y(x,t) = y_r(x-ct) + y_l(x+ct)$, where $c$ is the wave speed. For a vibrating string of length $L$, we have $c = 2f_0L$, where $f_0$ is the string fundamental frequency.

After the discretisation, the circuits contain two "waveguides" — delay lines corresponding to the two travelling waves. They are connected at both ends by filters, responsible for the wave reflection at both ends of the string (or the air column in the bore of a wind instrument). Because of the linearity of the delay lines and of typical filters, it is often possible to 'commute' some elements, e.g., to transpose a filter and a delay line, and to coalesce two delay lines into one. One should not forget that an open end of a wind instrument should be represented by a phase-inversion filter, since the outgoing low-pressure zone sucks air into the bore, producing a high-pressure incoming wave.

## 6 Digression: how to make noise

A typical random number generator is a 'stateful' entity, apparently not natural to code functionally, since it propagates the "seed" from one generation instance to another. But, if we want just to generate random *streams*, the algorithm is easy. We start with a simple congruential generator based on the fact that Clean operates upon standard 'machine' integers, and ignores overflows, which facilitates the operations modulo $2^{32}$. It returns a random value between $\pm 1$, and a new seed. For example:

```
rand1 seed
 # seed = 599479 + seed*25781083
 # r  = seed bitand 2147483647
 = (toReal r/2147483648.0,seed)
```

Despite its appearance, this is a purely functional construction, the **#** syntax is just a sequential variant of **let**, where the "reassignment" of a variable (here: **seed**) is just an introduction of a new variable, which for mnemotechnical purposes keeps the same name, screening the access to its previous, already useless instance. We write now

```
rndnoise seed
   # (z,seed) = rand1 seed
   = [#z : rndnoise seed]
```

This noise can be used in all typical manipulations, for example the "brown noise" which is the integral of the above "white noise" is just

```
brownnoise = z where   z=Dl (rndnoise someSeed + z)
```

In the generation of sounds the noise need not be perfect, the same samples can be reused, and if we need many streams it suffices to invoke several instances of **rndnoise** with different seeds. It is worth noticing that in order to generate noise in a purely functional way, *we don't need an iterative generator with propagating seed*!. There exist *pure* functions which behave *ergodically*, the values corresponding to neighbouring arguments are uncorrelated, they vary wildly, and finally they become statistically independent of their arguments. A static, non-recursive pure function

```
ergodic n
  # n = (n<<13) bitxor n   // Use let n'= ... if you don't like #
  = toReal (n*(n*n*599479+649657)+1376312589)/2147483648.0
```

mapped through the list $[0, 1, 2, 3, 4, \ldots]$ gives the result shown in Fig. 6. We have also used this variant of noise.

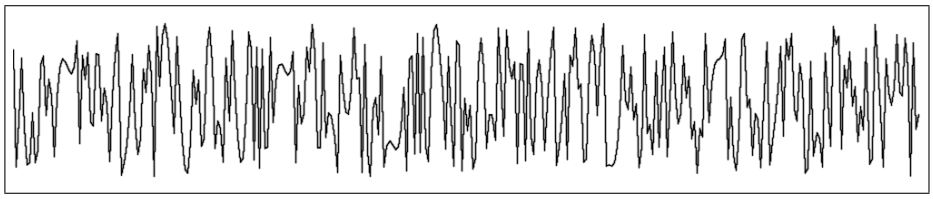

**Fig. 6.** An ergodic function

## 7 More general filters, and power series

We have seen that a simple recurrence, say $y_n = b_0 \cdot x_n - a_1 \cdot y_{n-1}$ is easy to represent as a stream (here: **y = b0*>x - a1*>Dl y**), but a general $m$-zero, $l$-pole filter: $y_n = \sum_{k=0}^{m} b_k \cdot x_{n-k} - \sum_{k=1}^{l} a_k \cdot y_{n-k}$ would require a clumsily looking loop.

However, both terms are just convolutions. The DSP theory [17] tells us that they become simple products when we pass to the $z$ transform of our sequences. Concretely, if we define $x(z) = \sum_{n=0}^{\infty} x_n z^{-n}$, and introduce appropriate power series for the sequences $b$, $a = [a_0 = 1, a_1, a_2, \ldots]$ and $y$, we obtain the equation $a(z) \cdot y(z) =$

$b(z) \cdot x(z)$, or $y(z) = H(z) \cdot x(z)$, where $H = b/a$. The stream-based arithmetic operations on formal power series are particularly compact [18]. Element-wise adding and subtracting is trivial, uses **`Map (+)`** and **`Map (-)`**. Denoting $x = x_0 + z^{-1}\bar{x}$, where $\bar{x}$ is the tail of the sequence, etc., it is easy to show that

$$w \equiv (w_0 + z^{-1}\bar{w}) = b \cdot x = (b_0 + z^{-1}\bar{b})(x_0 + z^{-1}\bar{x})\,, \tag{1}$$

reduces to the algorithm: $w_0 = b_0 x_0$, and $\bar{w} = b_0\bar{x} + \bar{b}x$.

The division $w = b/a$ is the solution for $[w_0 : \bar{w}]$ of the equation $b = a \cdot w$, and is given by the recurrence $w_0 = b_0/a_0$ and $\bar{w} = (\bar{b} - w_0\bar{a})/a$. So, the code for a general filtering transform is **`y = (b<*>x)</>a`** with

```
(<*>) [#b0:bq] a=:[#a0:aq] = [# b0*a0 : b0*>aq + bq<*>a]
(</>) [#b0:bq] a=:[#a0:aq] = [# w0 : (bq - w0*>aq)</>a]
      where w0 = b0/a0
```

It was not our ambition to include in the package the filter design utilities (see [10]), but since the numerator and the denominator of $H$ are lists equipped with the appropriate arithmetic, it is easy to reconstruct their coefficients from the positions of the zeros, by expansions. But, since the division of series involves the feedback, the programmer must control the stability of the development, must know that all poles should lie within the unit disk in order to prevent the explosive growth, and be aware that the finite precision of arithmetics contributes to the noise. But the numerical properties of our algorithms are independent of the functional coding, so we skip these issues.

## 8 More complex examples

### 8.1 Flute

One of numerous examples of instruments in the library STK of Perry Cook [11] is a simple flute, which must contain some amount of noise, the noise gain $g$ on the diagram depicted on fig. 7 is of about $0.04$. The flute is driven by a "flow", the breath strength,

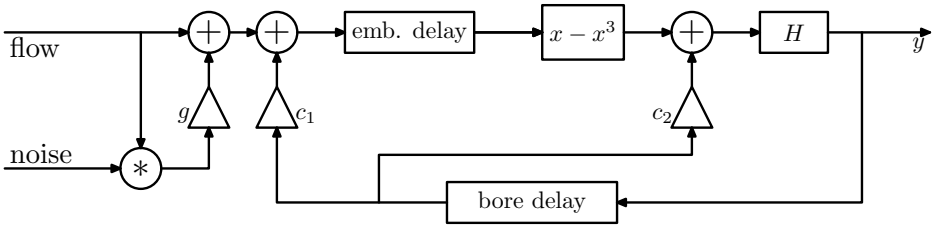


**Fig. 7.** A simple flute

which includes the envelope of a played note. There are two delays, the bore delay line, which determines the period of the resonance frequency, and a two times shorter

embouchure delay block. The filter $H$ is a reflection low-pass filter given by $y_n = 0.7x_n + 0.3y_{n-1}$. The gain coefficients are$c_1 = 0.5$, and $c_2 = 0.55$. The code is short.

```
flute freq flow = w where
 lpass1 x = y where y=0.7*>x + 0.3*>Dl y
 nlins xs = Map (\x -> x*(1.0 - x*x))xs
 u=0.04*>(flow*whitenoise) + flow
 v=u+0.5*>p
 p=tdelay (1.0/freq) w   // delay parameterized by real time
 w=lpass1 (0.55*>p + nlins (tdelay (0.5/freq) v))
```

An important ingredient of the instrument is the nonlinearity introduced by a cubic polynomial. Välimäki et al. [19] used a more complicated model. They used also an interpolated, fractional delay line, which is an interesting subject *per se*.

### 8.2 Bowed string

In Fig. 8 we see another example of a non-linearly driven, continuous sound instrument, a primitive "violin", with the string divided into two sections separated by the bow, whose movement pulls the string. Of course, after a fraction of second, the string slips, returns, and is caught back by the bow. The frequency is determined, as always, by the resonance depending on the string length.

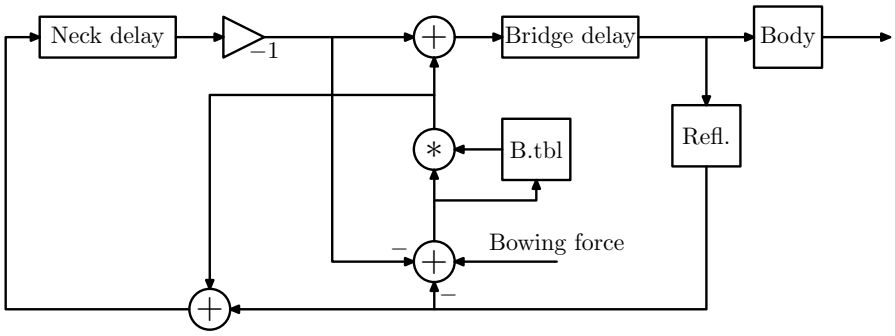


**Fig. 8.** A bowed string

```
bowed amp freq = y                 // amp is the bowing force
 where
  basedel = 2.0*SR/freq-4.0        // Base delay, split into neck and bridge
  p = 0.6 - 2205.0/SR              // Phenomenological filter pole position
  bowv = (0.03+0.2*amp)*>ones      // Max bow velocity; ones=[1,1,1,...]
  brefl= ~(bridge p 0.95 brdel)
  nrefl= ~(delay (toInt(0.872764*basedel)) (brefl+nvel))  // Neck
  vdiff = bowv-brefl-nvel          // Velocity difference; steering value
  nvel = vdiff*Map bowtable vdiff
  brdel= delay (toInt(0.127236*basedel)) (nrefl+nvel)
  y = biquad 500.0 0.85 0.6 brdel  // Output resonating filter
```

where **bridge** is the bridge reflection filter, and **bowtable** — the non-linearity (it could be table-driven, but here it is a local function). The function **biquad** is a filter. They are defined as in the source of STK:

```
bridge p g s = y where
  b0=(if(p>0.0) (1.0-p) (1.0+p))
  y = (g*b0)*>s + p*>Dl y

bowtable x
 # r=((abs(3.0*x)+0.75)^(-4.0))
 = if (r<1.0) r 1.0        // Conditional clipping
biquad freq rad g s = y    // Biquad resonating filter
 where
  a2=rad*rad
  v = (g*(0.5-0.5*a2))*>(s - Dl (Dl s))
  z = Dl y
  y = v + (2.0*rad*cos (freq*DpiSR))*>z - a2*>Dl z
```

Now, in order to have a minimum of realism, here, and also for many other instruments, the resonating frequency should be *precise*, which conflicts with the integer length of the delay line. We need a fractional delay. Moreover, it should be parameterized, in order to generate *vibrato* (or *glissando*); we know that the dynamic pitch changes are obtained by the continuous modification of the string length. Our stream-based framework, where the delay is obtained by a prefix inserted in a co-recursive definition seems very badly adapted to this kind of effects. We need more tools.

## 9 Fractional delay and time stretching

Some sound effects, such as chorus or flanging [20], are based on dynamical delay lines, with controllable delay. For good tuning, and to avoid some discretisation artefacts is desirable to have delay lines capable of providing a non integer (in sampling periods) delay times. Such fractional delay uses usually some interpolation, linear or better (e.g., Lagrange of 4-th order). The simplest linear interpolation delay of a stream **u** by a fraction **x** is of course **v=(1.0-x)*>u + x*>Dl u**, where the initial element is an artefact; it is reduced if we replace the 0 of **Dl** by the head of **u**. Smith proposes the usage of an interpolating all-pass filter, invented by Thiran [21], and co-recursively defined as

```
ifractd x s=:[#s0:_] = v + a*>u
 where
  a=(1.0-x)/(1.0+x)
  u = s - a*>v
  v = [#s0:u]
```

This module should be combined with a standard, integer delay. But how to change the big delay dynamically? The classical imperative solution uses a circular buffer

whose length (distance between the read- and write pointers) changes during the execution of the processing loop. In our case the delay line will become a stream processor. We begin with a simple, static time distortion. The function **`warp a x`** where $x$ is the input stream, and $a$ — a real parameter greater than $-1.0$, shrinks or expands the discrete "time" of the input stream. If $a = 0$, then $y_n = x_n$, otherwise $y_0 = x_0;\ y_1 = x_{1+a}; \ldots\ y_n = x_{n\cdot(1+a)}$, where an element with fractional index is linearly interpolated. For positive $\alpha$: $x_{n+\alpha} \equiv (1-\alpha)x_n + \alpha x_{n+1}$. Here is the code:

```
warp a [#x0 : xq] = [#x0 : wrp a x0 xq] where
  wrp g y0 ys=:[#y1:yq]|g>0.0 = wrp (g-1.0) y1 yq
                      =[#(1.0+g)*y1-g*y0 : wrp (g+a+1.0) y0 ys]
```

It is possible to vary the parameter $a$, the package contains a more complicated procedure, where the delay parameter $a$ is itself a stream, consumed synchronously with $x$. It can itself be a periodic function of time, produced by an appropriate oscillator. In such a way we can produce the *vibrato* effect, or, if its frequency is high enough – an instrument based on the frequency modulation. Moreover, if warp is inserted into a feedback, e.g., **`u=prefix ++| warp dx (filtered u)`** with a rather very small **`dx`**, the recycling of the warped **`u`** produces a clear *glissando* of the prefix.

The technique exploited here is fairly universal, and plenty of other DSP algorithms can be coded in such a way. If we want that a "normal" function **`f`** from reals to reals, and parameterized additionally by **`a`**, transform a stream **`x`**, we write simply **`Map (f a) x`**. Now, suppose that after every **`n`** basic samples the parameter should change. This may be interesting if together with the basic audio streams we have *control* streams, which vary at a much slower rate, like in Csound. We put the sequence of parameters into a stream **`as`**, and we construct a generalized mapping functional

```
gmap f s n x = wm 0 s x where
  wm k as=:[#a0:aq] x=:[#x0:xq]|k<n = [#f a0 x0:wm (k+1) as xq]
                                = wm 0 aq x
```

This may be used to modulate periodically the amplitude in order to generate the *tremolo* effect. A modified **`warp`** function, parameterized not by a constant but by a stream, if driven by an oscillator produces the *vibrato* effect. They are shown in Fig. 9.

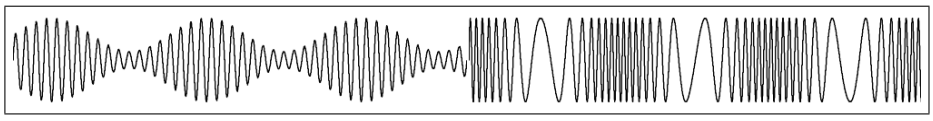

**Fig. 9.** Tremolo and vibrato

## 10 Reverberation

One of fundamental sound effects, which modifies the timbre of the sound is the composition of the original with a series of echos, which after some initial period during

which the individual components are discernible, degrade into a statistical, decaying noise. A “unit”, infinitely sharp sound represented by one vertical line in Fig. 10, is smeared into many. Fig. 10 corresponds to the model of John Chowning, based on the

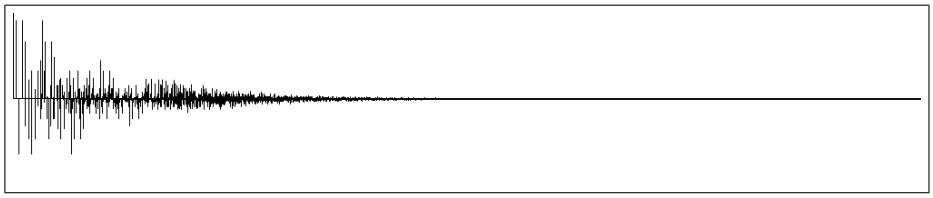

**Fig. 10.** Reverberation response

ideas of Schroeder [22], see also [23]. The reverberation circuit contains a series of all-pass filters as shown on fig. 4, with different delay parameters. This chain is linked to a parallel set of feed-forward comb filters, which are “halves” of the all-pass filters. They are responsible for the actual echo. Fig. 11 shows our reverberation circuit, whose code

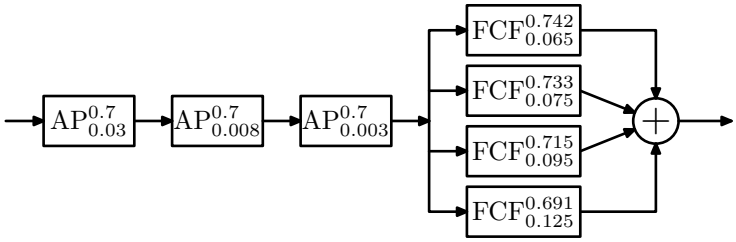


**Fig. 11.** Reverberation module

is given below. It is simple, what is worth noticing is a compact way of representing the splitting of a stream into components which undergo different transformations, as given by the auxiliary functional **`sumfan`**. The operator $o$ denotes the composition. Thanks to the curried syntax of Clean, the definition of a processing module composed out of elements linked serially and in parallel, doesn’t need the stream argument.

```
reverb =
 schr 0.03 o schr 0.008 o schr 0.003 o
   sumfan [fcf 0.065 0.742,fcf 0.075 0.733,
           fcf 0.095 0.715,fcf 0.125 0.691]
 where                                  // fcf is a forward comb filter
  fcf tm g x = x + g*>tdelay tm x       // Time, gain, stream

  schr tm = allpass tm 0.707            // Schröder allpass filter, below
```

```
sumfan l x
 # [l1:lq]=map (\f->f x) l
 = 0.25*>foldl (+) l1 lq
allpass tm b x = b*>z + v  where  // Co-recursive dispersion filter
  v = tdelay tm z
  z = x - b*>v
```

## 11 Conclusions and Perspectives

We have shown how to implement some useful algorithms for the generation and transformation of sound signals in a purely functional, declarative manner, using lazy streams which represented signals. The main advantage of the proposed framework is its compactness/simplicity and readability, making it a reasonable choice to *teach* sound processing algorithms, and to experiment with. We kept the abstraction level of the presentation rather low, avoiding excessive generality, although the package contains some other, high-level and parameterized modules. This is not (yet?) a full-fledged real-time generating library, we have produced rather short samples, and a decent user interface which would permit to pameterize and test several instruments in a way similar to STK [11] is under elaboration. The paper omits the creation of high-level objects: musical notes and phrases, not because it has not been done, or because it is not interesting, on the contrary. But this has been already treated in Haskore papers, and other rather *music*-oriented literature.

Our package can compute Fourier transforms, and contains other mathematical utilities, such as the complex number algebra, and a small power-series package. It has some graphic procedures (the signal plots presented in the paper have been generated with Clarion). We have coded some more instruments, as a clarinet, or a primitive harpsichord. Clarion can input and output `.wav` files, and, of course, it can play sounds, using the Windows specific `PlaySound` routine. We have generated streams up to 2000000 elements (about one minute of dynamically played sound at the sampling rate of 32000/sec.), and longer. In principle we might generate off-line sound files of much bigger lengths, but in order to play sound in real time it is necessary to split the audio stream in chunks. Our system uses heavily the dynamic memory allocation, and the garbage collection may become cumbersome. It is also known that the temporal efficiency of lazy programs (using the call-by-need protocol) is usually inferior to typical, call-by-value programs. We never tried to implement a full orchestra in Clarion…

Clarion remains for the moment rather a functional, but pedagogical feasibility study, than a replacement for Csound or STK, those systems are infinitely more rich. We wanted to show on a concrete example how the functional composition and lazy list processing paradigms may be used in practice, and we are satisfied. This work will continue.

## 12 Acknowledgements

I owe generous thanks to Henning Thielemann for interesting and useful comments, and for inspiring conversation.

## References


1. John William Strutt (Lord Rayleigh), *The Theory of Sound*, (1877).
2. Paul Hudak, Tom Makucevich, Syam Gadde, Bo Whong, *Haskore Music Notation — an Algebra of Music*, J. Func. Prog. **6**:3, (1996), pp. 465–483. Also: Paul Hudak, *Haskore Music Tutorial*, in: *Advanced Functional Programming* (1996), pp. 38–67. The system is maintained at: **`www.haskell.org/haskore/`**.
3. Richard Boulanger, *The Csound Book*, (2000). See also the site **`www.csounds.com`**
4. Julius O. Smith, Center for Computer Research in Music and Acoustics (CCRMA), Stanford University, Web publications at **`www-ccrma.stanford.edu/~jos/`**, (2003). Also: *Physical modelling using digital waveguides*, Computer Mus. Journal **16**, pp. 74–87, (1992).
5. J. Chowning, *The Synthesis of Complex Audio Spectra by Means of Frequency Modulation*, Journal of Audio Eng. Soc. **21**(7), (1973).
6. Kevin Karplus, A. Strong, *Digital Synthesis of Plucked Strings and Drum Timbres*, Computer Mus. Journal **7**:2, (1983), pp. 43–55. See also: D.A. Jaffe, J.O. Smith, *Extensions of the Karplus-Strong Plucked String Algorithm*, Comp. Mus. Journal **7**:2, (1983), pp. 56–69.
7. Bill Schottstaedt and others, *Common Lisp Music*, site **`ccrma-www.stanford.edu/software/clm/`**.
8. Roger B. Dannenberg, *Nyquist Reference Manual*, **`www-2.cs.cmu.edu/~rbd/doc/nyquist/`**.
9. Henning Thielemann, *Audio Processing Using Haskell*, Proc. 7th Int. Conf. on Digital Audio Effects (DAFx'04), Naples, pp. 201–206, (2004).
10. Matthew Donadio, HaskellDsp sources, site **`haskelldsp.sourceforge.net`**.
11. Perry Cook, site **`www-ccrma.stanford.edu/CCRMA /Software/STK/`**, see also the book: *Real Sound Synthesis for Interactive Applications*, A.K. Peters, (2002).
12. Rinus Plasmeijer, Marko van Eekelen, *Clean Language Report, version 2.1*, site **`www.cs.kun.nl/~clean/`**.
13. Oskari Tammelin, *Jeskola Buzz*, Modular software music studio, see e.g., **`www.buzzmachines.com/`**, or **`www.jeskola.net/`**.
14. Miller Puckette, *Pure Data: PD, Documentation*, **`crca.ucsd.edu/~msp/Pd_documentation/`**
15. François Déchelle, *Various IRCAM free software: jMax and OpenMusic*, Linux Audio Developers Meeting, Karlsruhe, (2003). See also **`freesoftware.ircam.fr/`**.
16. Ge Wang, Perry Cook, *ChucK: a Concurrent, On-the-fly, Audio Programming Language*, Intern. Comp. Music Conf., Singapore (2003). See also: **`chuck.cs.princeton.edu/`**.
17. Ken Steiglitz, *A DSP Primer: With Applications to Digital Audio and Computer Music*, Addison-Wesley, (1996).
18. Jerzy Karczmarczuk, *Generating power of Lazy Semantics*, Theor. Comp. Science **187**, (1997), pp. 203–219.
19. R. Hänninen, V. Välimäki, *An improved digital waveguide model of a flute with fractional delay filters*, Proc. Nordic Acoustical Meeting, Helsinki, (1996), pp. 437–444.
20. Sophocles Orphanidis, *Introduction to Signal Processing*, Prentice-Hall, (1995).
21. J.P. Thiran, Recursive digital filters with maximally flat group delay, IEEE Trans. Circuit Theiry 18 (6), Nov. 1971, pp. 659–664.
22. M.R. Schroeder, B.F. Logan, *Colorless artificial reverberation*, IRE Transactions, vol. AU-9, pp. 209–214, (1961).
23. J.O. Smith, *A new approach to digital reverberation using closed waveguide networks*, Proceedings, ICMC (1985).