Soundlab: Lazy Signal Combinators in Common Lisp

Max Rottenkolber

Sunday, 26 July 2015

Table of Contents

1 Introduction **1**

2 Rendering signals **2**

3 Signal synthesis **2**
3.1 Signals as functions 2
3.2 Signal combination 3

4 The state of Soundlab **6**

5 Conclusions **7**

6 Acknowledgments **7**

Written by Max Rottenkolber <max@mr.gy>, April 2013. Formatting has been updated since.

1 Introduction

The described approach is mainly inspired from experience gained by using analogue sound synthesizers. While every analogue synthesizer has its own unique sound based on the physical parts it is made of, most do share their key concepts. Usually a limited number of oscillators generate signals resembling—more or less—sine waves which are then modulated by being combined with each other in different ways.

Soundlab—an experimental implementation of the presented approach—is designed to enable the user to explore ways of signal combination. It does so by defining an embedded *domain specific language* which provides axioms that generate primitive signals and axioms that

combine arbitrary signals into new signals. The semantics of the language are based on a signal interface agreed on by every component. Furthermore Soundlab allows the use of *Common Lisp's* means of abstraction to define compound signals and signal combinators. Primitive as well as compound parts of the system form a homogeneous group of objects defined by their shared interfaces, which grant the system power and flexibility of a *Lisp* system.

There are of course many free software implementations (see for instance *Overtone* (`http://overtone.github.io`) and *Csound* (`http://www.csounds.com`)) of signal synthesis systems with programming language interfaces. Soundlab is—when compared to others—much simpler and entirely written and embedded in *Common Lisp*.

Soundlab is free software licensed under the *GNU AGPL* and can be obtained from *GitHub* (`https://github.com/eugeneia/soundlab`).

2 Rendering signals

Before discussing signal synthesis, we must define ways for consuming the synthesized signal as well as for verification of our results. Because our domain is music, we need to be able to play back signals as sound. Furthermore visualizing a signal can be useful for debugging since some properties of a signal are better conceived visually than aurally.

For both forms of presentation a technique called *sampling* is used—which will not be described in detail here. All that is needed to know for this approach, is that the sampling routine records a sequence of linear amplitude values according to a time span and a function—or signal—which maps values of time to values of amplitude. The resulting sequence resembles the kind of data that can be fed into standard digital sound adapters or plotting applications.

```
(function ((function (real) real) real)
          (sequence real))
```

Approximate type of a sampling function.

Soundlab derives its signal type from this rationale. It also exports two functions which record signals to standard *WAVE* audio files and *Gnuplot* compatible data files respectively. Soundlab also chooses arbitrary but sensible units and scales for time and amplitude. Time is chosen to be a number in seconds greater than zero and amplitude is chosen

to be a number ranging from -1 to 1. Results of inputs to the sampling routine exceeding these bounds are undefined.

3 Signal synthesis

3.1 Signals as functions

As discussed in the previous section, functions are the natural way to model a signal. Furthermore signals as functions encourage lazy operations without enforcing them—which can later be useful for aggressive optimizations.

```
(function (real) real)
```

Type of a signal.

A crucial type of signal is the sine wave—since in theory, all signals are sums of sine waves. *Common Lisp* provides us with a sine function `sin` which serves our purpose well. We could pass `#'sin` to a sampling routine as is, which would produce a very low frequency signal below the human hearing threshold. In order to specify other frequencies a constructor `sine` is defined which accepts a frequency in Hz and returns the respective sine signal.

```
(defun sine (frequency)
  (lambda (x) (sin (* 2 pi frequency x))))
```

Constructor for primitive sine signals.

Additionally a constructor for chorded signals could be defined as a function that takes two signals as arguments and returns a function that sums and normalizes them according to the boundaries we defined in the previous section.

```
(defun chord-2 (signal-1 signal-2)
  (lambda (x) (* (+ (funcall signal-1 x)
                    (funcall signal-2 x))
                 1/2)))
```

Constructor for a chord of two signals.

The `chord-2` function demonstrates the important traits of signals as functions. A new signal in form of an anonymous function is being

compiled whenever we call `chord-2`. Because the actual processing of the arguments is postponed until sampling occurs, operation on signals is cheap. Furthermore calls to `chord-2` can be combined to create chords with an arbitrary number of voices.

3.2 Signal combination

As seen in the previous section, modeling signals as functions enables us to write small, cheap and powerful signal combinators which can be chained to arbitrary extent. When chosen carefully, a small set of primitive combinators and signals can be used to create infinitely complex sounds.

```
(function (&rest (function (real) real))
          (function (real) real))
```

Type of a signal combinator.

While building `soundlab`, some primitives turned out to be especially useful. `flatline`—a constant signal constructor—serves a simple but important purpose. It takes a number as its only argument and returns a flat signal with a constant amplitude. When passed to a signal combinator its purpose is usually to scale combinations of signals. `add` is a general signal adder. It takes an arbitrary number of signals and sums them. Likewise, `multiply` multiplies signals. The `chord-2` combinator of the previous section can be defined more generally using these primitives.

```
(defun flatline (amplitude)
  (lambda (x)
    (declare (ignore x))
    amplitude))
```

Implementation of `flatline`.

```
(defun chord (&rest signals)
  (multiply (apply #'add signals)
            (flatline (/ 1 (length signals)))))
```

Generic implementation of `chord`.

Note that—due to the normalization performed by `chord-2`—the equivalent of `(chord a b c)` is

```
(chord-2 (chord-2 a b) (chord-2 c (flatline 1)))
```

as opposed to

```
(chord-2 (chord-2 a b) c)
```

which would produce the chord of `c` and the chord of `a` and `b` instead of the chord of `a`, `b` and `c`.

Furthermore, using signals as arguments to operations where constants would suffice whenever possible has proven to be feasible and powerful. Whenever a component is being modeled that would be controlled by a knob or fader in an analogue synthesizer, then its digital counterpart should be controlled by a signal. Take for instance a signal combinator `mix*` whose purpose is to merge two signals—just like `chord`—while additionally providing a way to control how much each input signal amounts to the mixed signal. So what would have been a *Dry/Wet* knob on an analogue synthesizer becomes a signal in our case. Our `mix*` takes three signals as arguments, two to be mixed and a third to control their amounts. For ease of implementation we also introduce `subtract`—the counterpart to `add`.

```
(defun mix* (signal-a signal-b ratio-signal)
  (add (multiply signal-a
                 (subtract (flatline 1)
                           ratio-signal))
       (multiply signal-b
                 ratio-signal)))
```

Implementation of `mix*`.

Staying within closure of the signal representation—that is trying hard to define our operations on a uniform signal representation only—grants the system a lot of power and flexibility. All of the presented signal combinators can be plugged into each other without restriction. As of now some care has to be taken to not produce signals exceeding the defined boundaries—see *Rendering signals*. Additionally, some combinators make use of non-audible signals. For instance `mix*` expects `ratio-signal` to return values ranging from zero to one and

`multiply` is used in combination with `flatline` to moderate signals. Soundlab fails to address the issue of having multiple informal subtypes of signals. As of now the user has to refer to the documentation of a combinator to find out if it expects certain constraints—as is the case with `mix*`. Nevertheless, our few examples can already be used to produce complex sounds. The code snippet below works in Soundlab as is and produces a rhythmically phasing sound.

```
(defun a-4 () (sine 440))
(defun a-5 () (sine 220))

;; Normalize a sine to 0-1 for use as RATIO-SIGNAL.
(defun sine-ratio ()
  (multiply (add (sine 1)
                 (flatline 1))
            (flatline 1/2)))

;; Produce a WAVE file.
(export-function-wave
  ;; A complex signal.
  (mix* (chord (a-4) (a-5))
        (multiply (a-4) (a-5))
        (sine-ratio))
  ;; Length of the sampling in seconds.
  4
  ;; Output file.
  #p"test.wav")
```

Possible usage of the presented combinators.

4 The state of Soundlab

As of the time of this writing Soundlab consists of roughly 500 lines of source code. It depends on a minimal library for writing *WAVE* files and is written entirely in *Common Lisp*. The source code is fairly well documented and frugal.

While being compact Soundlab provides basic routines for working with western notes and tempo, a few primitive waveforms, *ADSR* envelopes with customizable slopes and the ability to form arbitrary waveforms from envelopes, a good handfull of signal combinators and last

but not least an experimental lowpass filter. A stable API is nowhere near in sight but some trends in design are becoming clear.

On the roadmap are classic sound synthesis features like resonance, routines for importing signals from *WAVE* files and many small but essential details like bezier curved slopes for envelopes.

5 Conclusions

Soundlab—even in its immature state—presents an opportunity to explore abstract signal synthesis from scratch for engineers and artist alike. Its simplicity encourages hacking and eases understanding. While many complex problems surrounding signal synthesis remain unsolved, its lazy combinatorial approach forms a powerful and extensible framework capable of implementing classic as well as uncharted sound synthesis features.

The demonstrated approach proved to be especially suited to exploratory sound engineering. Ad-hoc signal pipelines can be built quickly in a declarative way, encouraging re-usability and creativity. In comparison to other tools in the domain the line between using and extending the system is blurry. Where *Csound* lets the user declaratively configure instruments and controls using *XML*, Soundlab emphasizes the user to use its built-in primitives and all of *Common Lisp* to stack layers of signal sources and modulators on top of each other. When compared to *Overtone*—a *Clojure* front-end to the *SuperCollider* audio system—Soundlab's back-end independency and simplicity make it seem more suited for exploration and hacking. Its core concepts are few and simple and its codebase is tiny and modular despite some advanced features like envelopes, musical scales and tempo, a lowpass filter and many kinds of signal combinators being implemented.

Many of *Common Lisp's* idioms proved to be an ideal fit for the domain of signal synthesis. Furthermore, embedding a signal synthesis language in *Common Lisp* provides the system with unmatched agility. While the core approach is mainly built on top of functional paradigms, extensions like signal subtype checking—as mentioned in section 3.2—could be implemented using macros.

I personally had tons of fun building and playing with Soundlab. I encourage everyone interested in computerized music to dive into the source code and experiment with the system—it really is that simple. Feedback and contributions are welcome!

6 Acknowledgments

Thanks to *Michael Falkenbach* (`http://soundpiloten.de`) for teaching me a whole lot about analogue audio hacking.

Thanks to Vera Schliefer for introducing me to signal theory.

Thanks to Drew Crampsie for providing the *Common Lisp* community with resources regarding the implementation of monadic combinators in the fom of *Smug* (`https://github.com/drewc/smug`).