Csound-expression Haskell framework for computer music

Anton Kholomiov (anton.kholomiov@gmail.com)

Abstract

The library csound-expression provides tools for sound design and electronic music composition. It embeds powerful audio programming language Csound in the Haskell. The library stays as close as possible to the pure functional programming. In the paper we are going to see how functional programming concepts can enhance creativity and reduce the complexity of music creation.

Introduction.

The electronic music is in bloom. There are thousands of virtual synthesizers. Almost all of them are visual tools. The programmers have rather limited choice of tools. We can mention SuperCollider, Csound, Overtone, ChucK, CommonMusic. The Csound is the oldest language in the list. It has the widest set of features (subtractive, granular, waveguide, sample-based synthesis, physical modeling, spectral processing, emulators for many analog synthesizer components). The community is active. Csound is implemented in C as a compiler and as a library. It opens doors to embedding it in all sorts of environments. It runs on all platforms (Linux, OS X, Windows), on devices (Raspberry Pi, Android, iOS) or even in the browser. But the language has old syntax and it's hard to code with Csound.

The csound-expression library is EDSL for Csound code generation. We can create the Csound code on the fly, invoke the Csound compiler on it and listen the result in the speakers. The library hides the Csound language behind the functional interface and tries to be as close to the Haskell way of thinking as possible.

In this paper we are going to know:

- The functional model for electronic music
- the main features of the library
- how functional programming can enhance the creation of electronic music

Functional model for sound design and computer music

In this section we are going to look at the model that describes music in the library.

Basic data types

Signals

The music is represented with audio signals or tuples of signals (in case of stereo or quad output). The signal is a stream of doubles or amplitude values. The type `Sig` represents signals in the library. We have numerical instances for signals. With signals we can create basic waveforms, that are typical for analogue synthesizers. Wave generators take in a frequency which is also signal and produce a static waveform that repeats itself with given frequency We can look at typical signature for pure sine wave, sawtooth, triangle and square waves:

```
osc, saw, tri, sqr :: Sig -> Sig
```

There are many more wave generators, we list the most frequently used ones. The filters can change the harmonic content, they take in a center frequency, resonance of the filter and signal to process. Here we can see low-pass high-pass, band-pass and band reject filters:

```
lp, hp, bp, br :: Sig -> Sig -> Sig -> Sig

lp centerFreq resonance asig = ...
```

When signal is created we can listen to it with function `dac` (it's short for digital to analog converter). Let's create an electronic music Hello World program:

```
> ghci
> :m +Csound.Base
> dac (osc 440)
```

If Csound is installed on our computer and sound card works properly we can hear a concert A 440 Hz played with pure sine wave.

Here we can find first severe digression from the Csound model. In Csound we have to declare a list of instruments and trigger them with notes. There are two sections called orchestra and score. But in the Haskell library we produce output with signals. There are no special sections. An instrument is a function from notes to signals and if we apply this function to score we get a simple signal as a result that we can feed to DAC. This model is more common for visual data-flow languages like Pure Data or Max MSP. Note that the signal which is created out of instrument application can become a part of another instrument. It greatly enhances the flexibility of composition process.

Constant values

There are special types for constant values: doubles (`D`), strings (`Str`), 1D arrays or tables (`Tab`). We can create an linear or exponential ADSR envelope generator:

```
leg, xeg :: D -> D -> D -> D -> Sig

leg attack decay sustain release = ...
```

A table can hold an array of doubles. Tables are often used for creation of periodic signals with custom shape:

```
oscBy :: Tab -> Sig -> Sig
oscBy waveShape frequency = ...
```

There are many functions to fill the tables with specific musical shapes: harmonic partials (`sines`), linear or exponential curves (`lins`, `exps`), splines (`splines`), Chebyshev polynomials (`chebs1`, `chebs2`) and many more.

The type `Str` represents Csound-strings they are often used to read audio files. We can create them out of haskell strings. Often sample reading functions already defined on Haskell strings:

```
text :: String -> Str
```

Spectrums

Csound has many functions for spectrum analysis. we can create spectrum (`Spec`) out of signal `Sig`, transform the spectrum and create the signal out of it:

```
toSpec :: Sig -> Spec
fromSpec :: Spec -> Sig
```

Recap: so we have stream values: signals `Sig` and spectrums `Spec` and constant values: doubles `D`, strings `Str`, tables `Tab`.

Side effects. IO-monad for csound

The side-effects are wrapped in the special type `SE`. It's an IO-monad for Csound. The Csound can have such side effects as generation of random values, saving audio to files, creation of references for mutable variables, updates of tables, creation of tables. For example we can create white or pink noise with functions:

```
white, pink :: SE Sig
```

The `SE` is also a monad so we can use a do-notation:

```
f cps = do
    wh <- white
    return (0.2 * wh + osc cps)
```

Instruments

An instrument is a function that transforms a tuple of constant values to tuple of signals:

```
instr :: (Arg msg, Sigs outs) => msg -> SE outs
```

The class `Arg` contains tuples of primitive constant types. The type `Sigs` contains tuples of signals or a single signal. Let's define an instrument:

```
instr :: (D, D) -> SE Sig
instr (amp, cps) = return (env * flt wave)
    where
        env = sig amp * leg 0.1 0.35 0.5 0.1
        flt x = mlp (freq * 0.5 + 1500 * env) 0.1 x
        wave = 0.3 * (saw freq + sqr (freq * cent 3))
        freq = sig cps
```

We have defined a simple instrument that consists of ADSR-amplifier (`leg`), moog low-pass filter (mlp) and a combo of sawtooth and square wave. The square wave is detuned by 3 cents. The filter has the same envelope as amplifier. The csounders might be surprised with things that haskellers can take for granted. Our instrument can be parametrized with shape of the wave or with type of filter. We can use functions as inputs:

```
genInstr :: (Sig -> Sig) -> (D, D) -> SE Sig
genInstr shape (amp, cps) = return (env * flt (shape (sig cps)))
```

It becomes very easy to try different shapes:

```
instr    = genInstr $ \freq -> 0.3 * (saw freq + sqr (freq * cent 3))
sqrInstr = genInstr sqr
```

Producing signals with instruments

Our instruments take in messages and produce signals. We can apply instruments to score (fixed list of notes), event streams and play them in real-time with midi-devices.

Midi-devices

The simplest way to try out ideas is to use midi-device. If you don't have one Csound provides virtual midi-keyboard (note that csound should be installed with GUI-support to provide this feature).

We can trigger an instrument right in the ghci:

```
> dac $ midi $ onMsg instr
```

We use function `midi` to trigger midi-instruments and function `onMsg` to convert simple instruments to midi instruments. While generic instruments take arbitrary tuples of constants midi-instruments expect messages to be in special format. The `onMsg` function reads amplitude and frequency from midi message and plugs it in our instrument.

To use virtual midi-keyboard we need to substitute `dac` with `vdac` (Virtual DAC). The great thing about the midi function is that it produces a signal as output.

```
midi :: Sigs a => (Msg -> SE a) -> SE a
```

It simplifies the process of application of effects dramatically compared to Csound. For example let's apply a reverberation:

```
> vdac $ fmap smallHall $ midi $ onMsg instr
```

The `smallHall` is just a function of the type `Sig -> (Sig, Sig)`. In the Csound we have to create a global variable to save the output of midi instrument and create a special always on instrument that reads that global variable applies reverberation to it, sends output to speakers and clears the global variable for the next step.

There is a shortcut called `mixAt` to specify dry/wet ratio of the FX:

```
> vdac $ mixAt 0.25 largeHall2 $ fmap fromMono $ midi $ onMsg instr
```

The `mixAt` applies an effect with given dry/wet ratio. We use stereo reverberation and convert our signal from mono to stereo with function `fromMono`.

Scores

Score is a list of values that are tagged with time stamps of start time and duration. So the note is a triplet of `(D, D, a)` where a is some message or tuple of constants. Actual implementation is a bit more complicated to allow fast composition of score. The Score API is heavily inspired by Paul Hudak's work on Haskore. The score datatype is defined in the separate package temporal-media.

We can create primitive score out of single value and rests (pause):

```
temp :: a -> Sco a
rest :: D -> Sco a
```

The temp creates a score that contains a single note that starts right away (has zero delay) and lasts for one second. We can transform score:

```
str, del :: D -> Sco a -> Sco a
```

The function `str` stretches the score in time domain. It scales all time stamps with given ratio. The function `del` delays all notes in the score y given amount.

We can combine the score:

```
har, mel :: [Sco a] -> Sco a
```

The `har` (short for harmony) plays the list os score as a chord (all start at the same time the duration of the result equals to duration of the longest score). The function `mel` plays one score after another. The total duration equals to the sum of durations for all notes. That's it! All basic unctions we need to create score. There is function to create loop of several segments:

```
loopBy :: Int -> Sco a -> Sco a
loopBy n a = mel (replicate n a)
```

It repeats the score n times. The score has instance for Functor, Traversable and Foldable!

Let's create score of notes:

```
> let notes = fmap temp [1, 5/4, 3/2, 2]
> let sc = str 0.5 $ fmap (\cps -> (0.3, 220 * cps)) $ mel [mel notes, har notes]
```

Let's create an instrument:

```
> let instr (amp, cps) = return (sig amp * leg 0.01 0.3 0.2 0.1 * tri (sig cps))
```

We can apply instrument to score with `sco` function and mix the score of signals to get the result:

```
> dac $ mix $ sco instr sc
```

Why do we need two step process of applying sco and then mix?

The `sco` function transforms the score of messages to score of signals to mix. It's like a tune played by single player in the orchestra:

```
sco :: (Arg a, Sigs b) => (a -> Sigs b) -> Sco a -> Sco (Mix b)
```

So we preserve the structure of score but notes now contain signals wrapped in the type `Mix`. By having this structure we can continue to build up a score. It's like arranging parts from several players. If the output of the score was a signal we couldn't arrange parts with higher level functions like harmony or melody composition. We can only treat signals as numbers the information on structure is lost. To render the arrangement from multiple instruments to signal we should apply the function `mix`:

```
mix :: Sigs a => Sco (Mix a) -> a
```

We can see how Haskell data types can provide higher order way of score construction. The csounders can only write the whole score note by note with absolute values for start time and duration of the note. This information locks the note blocks. To be fair Csound allows to define macro of skipping the time interval before note section starts to play this allows some support for relative time stamps. But it affects only limited support for melodical composition. We can not create blocks of notes. We can not store in the value score sections. We have to write notes line by line. The Haskell flexible data types simplify the process of score construction a lot.

Event streams

Event stream is a stream of messages or even score sections that happen at certain moments. Simple example of event stream is a sequence of metronome clicks. The clicks repeat at the certain rate. It would be nice to be able to trigger instrument or procedure when event happens. We can do it with the function:

```
sched :: (Arg a, Sigs b) => (a -> SE b) -> Evt (Sco a) -> b
```

The type of messages is very generic it contains score with the given type of messages. So we can trigger musical parts on every tick. The event stream is an instance of `Functor` type.

We can create the event streams with metronome:

```
> dac $ sched instr $ fmap (const $ temp (0.5, 220)) $ metro 1
```

We create a stream of clicks that happen one per second. Then we map the event stream messages to score that contain a single note and the apply an instrument to the stream and send the signal to speakers. The signal is going to play forever until we hit the Ctrl+C or shut the terminal.

There are many functions that let us transform the values in the stream. We can apply any state machine to the values. e can accumulate the state and transform values according to updated state. There are many useful special cases (see the docs for details). For example we can play a notes from the list at random:

```
> dac $ sched instr $ fmap (\x -> temp (0.5, 220 * x)) $ oneOf [1, 9/8, 5/4, 3/2, 2] $ metro 1
```

The `oneOf` is very generic function that let us pick values at random from the list when something happens:

```
oneOf :: Arg a => [a] -> Evt b -> Evt a
```

Also event stream is instance of `Monoid`. Empty value is forever silent stream and concatenation is a merging of messages from both sources. The event streams are also used to read user input from UI-widgets. We are going to look at how it woks soon.

Controlling instruments with UI

Csound has built in support for UI. It uses FLTK library. User can create buttons, sliders, knobs, rollers, checkboxes to control the audio output in real-time. While comparing to the modern standards the look and feel is very spartan it can be very handy an useful for prototyping a VST, creating an interactive performance or simply for tuning a parameter of the synthesizer with the virtual knob.

A UI element consists of two elements the visual representation (or `Gui`) and behavior. The behavior consists of three parts: output producer, value consumer and inner loop or state update:

```
type Widget a b = SE (Gui, Output a, Input b, Inner)

type Output a = a -> SE ()
type Input a = a
type Inner = SE ()
```

There are special cases:

```
type Sink a = SE (Gui, Output a)
type Source a = SE (Gui, Input a)
type Display = SE Gui
```

The widget is not only a single widget it can contain a whole section of sub-widgets. We can combine Guis with simple functions:

```
hor, ver :: [Gui] -> Gui
sca :: Double -> Gui -> Gui
space :: Gui
```

A visual representation of the widget is some picture that is drawn inside a rectangle. With layout functions we can organize the rectangles on the screen. The functions `hor` and `ver` stack rectangles horizontally or vertically, the `sca` (short for scale) expands or shrinks the relative size by the given ratio. The `space` creates an empty space. So user don't have to care for specific sizes all sizes are relative. We only need to set up the minimum amount of relative sizes.

We can combine behaviors by ordinary functions. There are constructors that can create widgets:

```
sink    :: SE (Gui, Output a) -> Sink a
source  :: SE (Gui, Input a) -> Source a
display :: SE Gui -> Display
```

Here is how we can create the signal that is controlled by two knobs:

```
main = dac $ source $ do
    (ga, amp) <- uknob 0.5
    (gb, cps) <- xknob (50, 1000) 220
    let g = hor [ga, gb]
    return (g, amp * osc cps)
```

The `uknob` creates a knob that outputs unipolar linear value that ranges in the interval [0, 1]. Its single argument is the initial value. The `xknob` creates an exponential value that ranges within specified interval (exponential scale is better suited for frequencies). The function `dac` can render values of many different types. Also it can render the signals that are produced with UI.

The real achievement of this model is that the result of combination is also widget and we can use it as a black box and combine it with other widgets.

Applicative style combinators for sources

There are special combinators that combine visuals and behaviors at the same time:

```
lift1 :: (a -> b) -> Source a -> Source b

hlift2, vlift2 :: (a -> b -> c) -> Source a -> Source b -> Source c

hlift3, vlift3 :: (a -> b -> c -> d) -> Source a -> Source b -> Source c -> Source d
```

The `hlift2` function applies the function to the outputs of two sources and stacks visuals horizontally. The `vlift2` function makes the same thing only it stacks visuals vertically.

There is a function that combines lists of sources:

```
hlifts, vlifts :: ([a] -> b) -> [Source a] -> Source b
```

We can redefine previous example with much more succinct expression:

```
main = dac $ hlift2 (\amp cps -> amp * osc cps) (uknob 0.5) (xknob (100, 1000) 220)
```

Events and GUIs

The sliders and knobs produce signals but for discrete widgets like buttons and knobs this representation is not so convenient. The discrete output is represented with event streams. Here is the signature of the button:

```
button :: String -> Source (Evt Unit)
```

It takes in a label that is going to be written on the button's face. It produces the event stream of clicks. and we can use all the library functions that are defined for event streams. Here we can play a note when user presses the button:

```
> let f evt = sched instr $ fmap (\x -> str 0.2 $ temp (0.5, 220)) evt
> dac $ lift1 f (button "play")
```

Also we can merge the event streams from several UI-sources with Monoid instance transform the event streams with library functions, execute procedures.

Standard library of functions

Almost all csound audio units are implemented in the library. But some of them tend to induce the Csound style of programming. The module `Csound.Air` defined many functions that promote the Haskell way of thinking. They reduce the number of arguments and reverse the order of arguments for point-free style. Many units have more intuitive interface. Granular synthesis and hypervectorial synthesis were redesigned to provide sensible defaults for many parameters so that user can supply them only when he wishes to do so. For example the function partikkel has more than 20 arguments. It's inevitable in Csound since Csound doesn't have any means for grouping data structures. But in Haskell we can group all rarely used parameters in the record and provide the default value for the user. The partikkel is a wonderful function but the need for specification of all parameters stops many users from unlocking it's real powers. The haskell version of it has only 5 parameters and one for record that contains the defaults.

If you don't know were to start to learn the library it's better to start with module `Csound.Air`. First check out the waves, filters and envelopes. This will give you enough tools to delve into subtractive synthesis.

Standard library of instruments

Almost all VST and hardware synthesizers come with the library of patches. The csound-expression also provides a user many instruments to play out of the box. They are defined in the package csound-catalog. The Patch is an instrument and the chain of effects (see `Csound.Air.Patch`). There are functions that can apply patch to midi-device input, event streams, score. The csound-catalog has many beautiful instruments defined by fellow csounders and recoded in the haskell. We can find the full list at the module `Csound.Patch`.

Functional model for composition

The most important concept of the library is modularity. Modularity means that an object can be self contained and used as a black box. There are functions that can combine those boxes together and the result only depends on the arguments.

For instance the result of application of an instrument to an event stream is just a signal. Which we can use almost everywhere in the library. It gives us great flexibility and enhances the expressiveness. The modularity also makes things reusable. We can define a module of instruments. And our fellow musicians can use them just by importing.

The mere ability to assign the anything to variable and pass it to a function and receive from a function empowers csound with expressiveness that's hard to imagine within the scopes of csound. To be fair for Csound

it excels at audio units. Many units are on par with commercial analogs and sound quality can be supreme. Interesting strength of Csound is ability to work in off-line mode. If we have a very complex audio generator we can wait for it to render the sound offline. That's not possible in Pure Data or other real-time environments. Csound often doesn't sacrifice the quality for speed.

The Csound has many global parameters that prevents things from being modular the copy and paste approach becomes ubiquitous. But in the library we hide those details behind composable APIs. Good example of re-usability is the package `csound-catalog`. It defines many standard instruments. That can be used right out of the box. If you need to play an organ emulator you can load the modules `Csound.Base` and `Csound.Patch` in the ghci and type:

```
> dac (atMidi toneWheelOrgan)
```

and you are ready to perform. You can tweak the defaults if you need but for quick sketches the default behavior should just work.

Related work

The closest project to csound-expression is wonderful Overtone library. The Overtone ports the SuperCollider to Clojure and adds the ability for live-coding. The csound-expression doesn't provide livecoding. It's write and compile approach. But csound-expression provides more audio units, it also provides collection of instruments that are ready to play.

There are Haskell projects to create electronic music. The Tidal provides the user DSL for creation of musical patterns and jamming live. The DSL creates a stream o control messages to a sampler Dirt that ships with Tidal. The csound-expression gives the user ability to create the synthesizers and samplers. The focus of tidal is creation of performances with the set of samples but csound-expression covers the full stack from sound design to creation of music and performances though it doesn't provide livecoding.

Haskore and Euterpea have main focus on score creation. There is support for Csound in them but it's not mature and sketches a dozen of functions that can be combined in Csound way. So there is no functional model on top of Csound AST. It feels like building AST with haskell but the csound-expression provides it's own functional model for electronic music and sound design which hides many imperative details.

Haskell package synthesizer focuses on sound design. it implements many audio units with Haskell. It can be used with Haskore to create music. The csound-expression is based on Csound and it implements more audio units and various synthesis techniques. This is the strength of the library that it's build on top of solid foundation.

Conclusion

The Csound is one of the most powerful environments for creating digital audio. It has very old syntax which prevents many user from unlocking it's real powers. The library csound-expression embeds Csound in Haskell. Moreover it provides not just construction of Csound AST with Haskell. It has it's own model for representation of sound design and music composition domains. Many idioms are borrowed from Haskell libraries (like managing side effects with monads, expressing UIs with event streams).

But there is another side of the moon. The csound-expression is the big project and so far it has only one developer which is not a good thing. Not all modules are properly tested but the functionality is very reliable and stable thanks to solid testing built in Haskell (like static type checking). The csound-exprression was used on android, custom synthesizer and even on stage! you can listen to the music that was created with the library on soundcloud [2]. The synthesizers from standard collection were used with midi-keyboard on several festivals of world music and electronic music (Trimurti, Solar Systo).

We hope that the library could introduce you to the world of text-based sound design and music production and let you create musical tracks more easily.

[1] Github repo for csound-expression: https://github.com/spell-music/csound-expression

[2] Musical examples: https://soundcloud.com/anton-kho/