CSOUND ACCELERATED WITH HASKELL

INTRODUCING THE LIBRARY CSOUND-EXPRESSION

Anton Kholomiov
anton.kholomiov@gmail.com

Csound is a very powerful audio engine, but by syntactic ease of use it is still far behind the modern languages like Python, Ruby or Haskell. That is sad because clumsy syntax prevents many users from unlocking the real powers of Csound. In my library csound-expresion I'd like to take the best parts of Csound (powerful audio engine and scheduler) and supply them with the wings of syntax gifted to Haskell. The proposed solution can greatly enhance the productivity of the Csound musician.

Introduction

The csound-expression is a code generator. It produces Csound code out of Haskell code. This paper shows what benefits the user can get by using Csound with Haskell:

- Interactive music production. Imagine that you can type an opcode in the interpreter, apply a filter to it, add some reverberation, hit enter and you can hear the sound right away!

- Everything is an expression. There are no special forms in the language like orchestra, scores, and opcodes. All entities are represented with elementary units like value or function. For example, an instrument is a function that takes in notes and produces signals. If we want to trigger the instrument with notes we can apply a function to the instrument and notes and we get a signal as an output. The result can become an argument for another function. No need for identifiers for instruments or ftables. The framework generates them behind the scenes.

- Palette of audio units for many common techniques. Many opcodes were redesigned in more convenient form. For example there are basic band-limited waveforms of analog synthesizer (osc, saw, sqr, tri) the granular synthesis opcodes were simplified by setting default values for many arguments (user can tweak the defaults) etc.

- Flexible functions. The functions are values. A function can take in another function as an argument and produce the functions as a result. Imagine that your opcode can take in a filter as a parameter. It's not a special integer argument like 0 for low pass, 1 for high pass. It takes a function that represents the filter; you can pass even a chain of filters as long as it takes the same arguments as a simple filter.

- Data structures. The Csound is embedded in the Haskell. So it inherits all Haskell's data structures. We can use lists, trees, maps; create our own data structures with fields.
- Modularity. We can define a set of audio units. We can put them in the library and we can share it with our friends as a library. So the client don't need to copy and paste the definition. User can just install our library and import the definition of the wonderful effect we made.

- Functional Reactive Programming. It's a great way of representing computations that depend on event streams and scheduling. The core idea is that instead of listening for the events and appending callbacks to the listener we can create the values that contain all future events and treat them as containers of values like lists. We can map a function over a stream. We can filter the values of the stream with a predicate. We can merge a couple of streams together and so on.

- Easy to use GUIs. The GUI widget is just a function that produces a visual representation and the value (signal or event stream). We can combine visuals and values with another functions.

- Code optimization. The code is optimized so that common sub-expressions are eliminated. The variables for signals are reused as much as possible. So that an instrument takes minimal chunk of RAM to run.

- The output signal is limited by amplitude to protect your ears and speakers from damage.

The rest of the paper describes some of the topics in depth. Please read the Appendix if you don't know the Haskell. You can get the basics of the syntax there.

1 Main data types

In csound-exression there are six primitive data types: `D` for constant numbers, `Sig` for signals (control or audio), `Str` for Strings, `Tab` for ftables and `Spec` for spectra (used in pvs-opcodes).Also there are two boolean types: `BoolD` for constant booleans and `BoolSig` for boolean signals. The stereo signal is a tuple of signals. The type of the signal is derived from the context. If user plugs the signal into k-rate variable it becomes a control signal the same is true for audio rates.

2 REPL. Interactive music production

Many modern languages have the REPL or read-eval-print loop. It's like an interactive mode in the Python. We can type an expression, hit Enter and we can see the result. We

can use it to create the sound right in the interpreter. The function dac sends the output to speakers:

```
> dac (osc 440)
```

The number of outputs is derived from the context. Here is the stereo output:

```
> dac (saw 220, tri 220)
```

Notice that there is no rate information no globals are set. There are sensible defaults suitable for real-time usage. If we want to alter the defaults we can use the function dacBy. It takes the options as a first argument:

```
> let run asig = dacBy (setRates 48000 10 <> setDac "hw:0,3") asig

> run (loopWav 1 "drums.wav")
```

There is a binary operator <> which constructs options from primitive parts. Here we set the audio and control rates and the name for dac output channel.

3 Everything is an expression

The main strength of the library comes with a simple motto:

Everything is an expression

Every musical concept that we are going to create is either value or function. So the musical idea belongs not to the language but to the values of the language.

We can manipulate musical ideas with tools that language provides. Let me clarify the difference with an example. In the Csound we have two main sections: orchestra and score. We can place them in the same file by using XML-elements. Instruments and tables can interact with each other by numbers or special ids or names that we declare. Those parts belong to the syntax of the language. It's fixed. We can not manipulate XML-elements with Csound code. We can not give a name to the bunch of notes or create an instrument inside another instrument. But what if we could do it? In the Haskell library an instrument is a function:

```
mySynth :: (Arg a, Sigs b) => a -> SE b
```

The `Arg a` means that arguments of the function should belong to the predefined set of types. They are all the types that Csound can store in p-fields (`D`, `Str`, `Tab` and tuples of them). The `Sigs b` means that output of the function should be a tuple of signals or just a signal. It's a Haskell way to restrict a set of types. The prefix `SE` in the result means that an instrument can produce side effects (side effect is random number generation, allocation of memory for delay lines, writing to files).

Let's create an instrument:

```
pureSine :: (D, D) -> SE Sig
pureSine (amp, cps) = return (sig amp * osc (sig cps))
```

The instrument expects a pair of arguments: amplitude and frequency for the sine wave (`osc`). There is a difference between Haskell and Csound that Haskell that automatic conversions are not allowed. We need to convert the numbers to signals with function `sig`. The `osc` expects a signal as frequency. Let's trigger the instrument with notes. In Haskell to trigger an instrument we call a function:

```
> let notes = str 3 (mel (fmap temp [(1, 220), (1, 330), (1, 440)]))

> dac (mix (sco pureSine notes))
```

The `sco` function takes an instrument and a bunch of notes.

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

The `Sco` is a type that looks very much like list. But with `Sco` each element of the list has two timestamps (for start time and duration). With `Sco` we can assemble the notes from primitive parts. We don't need to write down the whole list as in Csound. We can use functions to combine notes.

The mix function converts the value of type Sco to signal

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

Right now it's not that important to understand how the list of notes is constructed. The important thing to note is that the list of notes is a value. We can give it a name, we can store it in the list, we can create a function that takes the list of notes in and produces some result. In the first line we create three notes:

```
> let notes = str 3 (mel (fmap temp [(1, 220), (1, 330), (1, 440)]))
```

The function `temp` creates a note that lasts for 1 second and has no delay. The fmap applies a function temp to all elements in the list. The function `mel` takes a list of scores and produce a single score so that all scores in the list are played sequentially (one after another). The next function `str` (short for `stretch`) rescales the time stamps so that each note lasts for 3 seconds.

The great part of it is that we can use all entities as simple values. We can generate the notes from the function. We can store the instruments in the list:

```
let q = [pureSine, sawSynth, triangleSynth]
```

Moreover the result of expression is just a signal. We can use it in another instrument. That's how we can add a reverb to the signal:

```
> let asig = mix (sco pureSine notes)

> dac ((asig, asig) + 0.4 * (smallHall asig))
```

The function `smallHall` produces a stereo reverb form the mono input. So to add the original a signal to it we need to convert it to stereo. There is no need for global variables to make a reverb. The value `asig` contains the whole signal within it. We can apply another effect to it. It can become a part of another instrument! The plus sign is overloaded for tuples of signals. So we can add stereo signals.

The same thing applies to many other concepts. We are going to look at them in the later sections. We construct values for GUIs, for event steams. We can store GUI widgets in lists, combine them together and give the name to the combined widget or we can create a function that is parameterized with a GUI-widget.

4 Data structures

It's hard to cover all data structures in the short paper. So we are going to look at the couple of examples. Let's look at some benefits of being able to use lists. That's how we can create an audio unit for additive synthesis:

```
additive :: (Sig -> Sig) -> Int -> [Sig] -> Sig -> Sig

additive f size amps cps = sum (fmap makeHarmonic (zip amps [1 .. size]))

      where makeHarmonic (amp, n) = amp * f (sig (int n) * cps)
```

We create a range from 1 to the given integer: `[1 .. size]` Then we zip it with the list of amplitudes: `zip amps [1 .. size]`. The function zip takes two lists and creates a list of pairs. Then we transform all the values with function `makeHarmonic`. It creates a single harmonic. The function sum sums all values. That's it! Two lines of code! Imagine how you can write it in the Csound language. Let's apply this function:

```
> dac (additive tri 4 [1, 0, 0.4, 0.1] 440)
```

5 Modularity

The Haskell has support for modules. We can define our set of instruments, default values and handy functions. Then we can create a library out of it and redistribute. We can send the library to our friend. The user don't need to copy the UDOs, the user can install the library and use it with import statement:

```
import Csound.Stuff(fabulousSynth)
```

Our friend is ready to use the synthesizer we've made. We can create not only instruments but also new frameworks! We can use the basic building blocks and define our own ways of organizing music and workflow. Let's consider an example. The csound-sampler is a library that is built on top of csound-expression. It targets the creation of music out of small samples and patterns of samples. The main idea of the library is to supply the signal with BPM-rate and duration. We combine the samples so that they are aligned by BPM-rate. We can delay the sample within the grid; loop over samples limit the length, sequence to play one segment after another and so on.

That's how we can load three samples and play them in sequence with the drum loop:

```
> dac (sum [ mel (fmap wav ["fox1.wav", "fox2.wav", "fox3.wav"])

      , loop (wav "drums.wav")])
```

6 Functional Reactive Programming (FRP)

FRP is a great way of representing computations that depend on event streams and scheduling. The main idea is that instead of listening for the events and appending callbacks to the listener we can create the values that contain all events and treat them as simple containers of values like lists. We can map a function over a stream. We can filter the values of the stream with a predicate. We can merge a couple of streams together. We can accumulate some value with events from the stream and so on.

Event streams

The event stream is a function. It takes in a procedure that expects content of an event as argument. It takes in a single event and performs some action. The event stream performs the procedure every time the event is fired. With function `metro` we can construct an event stream of empty tuples. The frequency of event productions equals 1 per second: `let evts = metro 1`

There are many functions that work with event streams as if they are lists. We can transform the elements of the event stream: `fmap :: (a -> b) -> Evt a -> Evt b`

We can count the number of events so far: `appendE 0 (+) (fmap (const 1) (metro 1))`

The function `appendE` takes in an initial value to update and the function of two arguments. when an event happens it applies the function to the current value of the state and to the value of the event and puts the result in the state and in the output stream.

We can merge values from two streams together with binary operator `<>`: `metro 1 <> metro 3`

The output stream contains the events from both streams. Once we have events we can trigger instruments with them or convert them to signals:

```
evtToSig :: D -> Evt D -> Sig
```

The `evtToSig` converts an event stream to signal. The first value is held while nothing happens.

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

The `sched` triggers the instrument with event stream. The instrument is just a function: `(a -> SE b)`. The output is the signal. It can be used in another instrument right away! The necessary boilerplate code is generated behind the scenes.

Graphical User Interfaces (GUIs)

Lots of effort was put into making the GUIs easy to use. The main idea is that a widget is a pair of values. The first is for visual representation and the second one is for the actual value that widget produces (it can be a signal in case of knob or an event stream

in case of button). There is special function to combine visuals in an easy way. User has no need to specify exact values for positions and sides of the bounding box of the widget; the system tries to calculate it for the user.

There are basic functions:

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

sca :: Double -> Gui -> Gui
```

The `hor` and `ver` aligns horizontally or vertically a list of widgets. So each widget takes an equal amount of space. We can alter the amount of space by scaling the widget with function `sca`. We can leave the empty space in place of widget with `space`.

That's how we can create two knobs and align them horizontally:

```
main = dac $ do

    (gui1, cps) <- uknob 0.5

    (gui2, q) <- uknob 0.5

    panel (hor [gui1, gui2])

    return (lp (500 + 1500 * cps) (5 + 20 * q) (saw 110))
```

We create two unipolar knobs (ranges between 0 and 1) with initial values 0.5. Then we create a panel that contains the visuals. At the last statement we produce the result. It's a filtered sawtooth wave. The parameters for filtering are updated with knobs.

The example can be further simplified. There are functions that combine the values produced by the widgets and the visuals for the widgets at the same time:

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

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

Let's take the `hlift2` as example. It expects a function of two arguments and two widgets that produce values (or `Source`s). The function is applied to values and the visuals are stacked horizontally. The result of the function is itself a widget. So it can be passed to another function `hliftN`!

Let's rewrite the example:

```
> dac (hlift2 (\cps q -> lp (500 + 1500 * cps) (5 + 20 * q) (saw 110))
(uknob 0.5) (uknob 0.5))
```

In this variant the panel is applied automatically. So we get a single expression that takes values from two knobs and creates a filtered sawtooth wave which depends on the values. It's a single line of code! You can try to rewrite it in Csound. There are many more widgets beside knobs.

Let's study another example. Let's create two buttons one is going to raise the value by one and another should lower the value by one. We create a signal out of this value then

we turn the value to frequency and apply a triangle wave to it. Lets' create two buttons first:

```
> let btnUp = button "Up"

> let btnDown = button "Down"
```

Then we create a function that takes in two streams, substitutes the values of events to 1's in the first stream and to (-1)'s in the second one. We merge two streams together with function `<>`. We create running sum for each coming event (`appendE (-1) (+)`) and we add 60 to all values to get midi notes (`fmap (+ 60)`). The function stepper creates a sample and hold signal for event stream.

```
> let toMidi ups downs = stepper 0 (fmap (+ 60) (appendE (-1) (+)

      (devt 1 ups <> devt (-1) downs)))
```

In the last line we apply the function toMidi to both streams and channel the signal of midi-keys to the triangle wave:

```
> dac (vlift2 (\ups downs -> fmap (tri . sig . cpsmidinn) (toMidi ups
downs) btnUp btnDown))
```

We use `vlift2` to align the visuals vertically. The really great part of it that the result is a widget itself! Let's create a chord of three notes. It's as simple as giving a name and summing the result of all notes:

```
> let note = vlift2 (\ups downs -> fmap (tri . sig . cpsmidinn)
(toMidi ups downs) btnUp btnDown)

> dac (hlift3 (\a b c -> mul 0.4 (mean [a, b, c])) note note note)
```

Conclusion

The proposed library embeds Csound in Haskell. With Haskell we get flexible functions and data structures. The library is designed so that musical concepts are turned into values and functions. We can create complex musical structures out of simple primitives.

The library is open source (hosted on github). It's freely available on Hackage (http://hackage.haskell.org/package/csound-expression). Examples of the music created with it can be found on soundcloud: https://soundcloud.com/anton-kho.The only price the user have to pay is the price of learning new programming language. This journey is not for everybody. The user has to decide is it worth the effort. But the code reduction rate can be as high as 100 lines of Csound code for one line of Haskell code. We can get wonderful and free tutorial for Haskell at http://learnyouahaskell.com/. The library was designed to be as simple as it can be. I was on the guard against the brainy concepts that may scare an artist.

Appendix: drops of Haskell syntax

I assume that the reader is not versed in Haskell, but lots of Csounders know Python so there is a transcription to Python.

Function signature: `f :: ArgType1 -> ArgType2 -> ArgType3 -> OutType`

Function definition: f `a1 a2 a3 = …` Python: `def f(arg1, arg2, arg3):`

Function application. In Haskell we use white space as delimiter. Application of the function the function's name goes first the rest names are arguments:

Haskell	Python
`f a1 a2 a3`	`f(a1, a2, a3)`
`g (f a) b (h x)`	`g(f(a), b, h(x))`

Application without brackets: `f a $ g b === f a (g b)`. In Python: `f(a, g(b))`

We put a dollar if we want to enclose all following symbols in parenthesis. Local arguments:

```
f a = g a b

      where b = expr1
```

Acknowledgements

I'd like to mention those who supported me a lot with their music and ideas:

Music: entertainment for the braindead, three pandas and the moon, odno no, Hariprasad Chaurasia, annsannat & alizbar, toe, iamthemorning, atoms for piece / radiohead, Dolphin, loscil, boards of canada, Four Tet, Hozan Yamamoto, Tony Scott and Shinichi Yuize (music for zen meditation album).

Ideas: Conal Elliott (for FRP ideas), Oleg Kiselyov (for implementation of CSE algorithm), Paul Hudak (for representation of scores), Gabriel Gonzalez (for FRP implementation), Rich Hickey (on how to think and solve problems) Iain McCurdy (for wonderful Csound instruments) and Csound's community.