Haskell meets Csound

Playing with Csound in functional programming language

Anton Kholomiov
Author AT anton.kholomiov@gmail.com

Abstract

This paper describes how features of functional programming language like Haskell can enrich Csound syntax and presents an implementation of proposed ideas in csound-expression, Haskell library and Csound code generator.

The music composition can be seen as a process of patterns creation, ordering and transformation of little pieces. Such a natural features of functional programming as higher order functions and algebraic data types are well suited for the task of creating complex patterns by means of composition and transformation.

I. Introduction

I'd like to introduce you to csound-expression (CE). It is a Haskell library and Csound code generator. Csound is a language. Why should anyone generate its code? Why not to use Csound directly? Csound is a very simple language. It was designed to be learned quickly by artists. But this simplicity can be a real hindrance. Csound code tends to become verbose and low-level. In this article we are looking for structures to speed up Csound syntax.

In CE musical ideas are expressed with composition of functions. It's like growing a tree. You can see Csound units in place of the leaves and Haskell functions in place of the branches. Csound has a very powerful set of units for sound synthesis. But it is difficult to compose these units. Haskell is a functional language with strong static typing. It supports modularity, higher order functions, currying, algebraic data types and type classes or powerful tools for abstraction in short. It can help Csound to compose things together.

Section II describes why functional programming is good for Csound. Section III provides you with overview of CE library. CE is evolving project, so this section shows you some drawbacks of the library alongside with its advantages. Section IV compares CE with other similar projects and pictures some future plans.

II. Why functional programming language?

Csound handles sound synthesis (orchestra) and score composition (score). In this section we'll see how functional programming (FP) can facilitate both of them.

Orchestra

Orchestral Csound is a modular synthesizer. Modular synthesizers follow the same pattern. There is a set of units, each of them can generate a sound or transform it in some way. A unit has inputs and outputs and we can link units together with cords. Csound allows us to do it with text. A specific combination of units and linking cords is usually called a patch. A patch is a pattern of linked units.

From the functional point of view units are functions and linking with cords is an application. Each cord has direction. It links output of one unit with input of another one and allows signal to flow from one unit to another one. So a patch is a function which is composed from another functions.

Csound provides a user with two constructs of patch building. First is linking units with cords by means of variables and second is user defined opcodes. With user defined opcodes you can give a name to some patch (or function). By giving names you can save your typing efforts and underline a structure of your patch. A user defined opcode has inputs and outputs. It's like predefined unit. FP can take it further. FP supports higher order functions, i.e. functions that can take functions as input or produce functions as output. Let's look at simple example. Fourier series:

$$f(w,a)=a_0(t)+a_1(t_i)\cdot\cos(1wt_i+a)+a_2(t_i)\cdot\cos(2wt_i+a)+...+a_n(t_i)\cdot\cos(nwt_i+a)$$

$$t_i=h\cdot i,i=1..\mathrm{N}$$

The function f produces signal. Suppose we want to use another basis function, then if our language supports higher order functions we can define function that takes in a basis function.

$$g(e,w)=a_0(t_i)+a_1(wt_i)\cdot e(1wt_i)+a_2(t_i)\cdot e(2wt_i)+...+a_n(t_i)\cdot e(nwt_i)$$

And then

$$h(a)(x)=\cos(x+a)$$

$$f(w,a)=g(h(a),w)$$

A filter function can become a parameter of your patch. And this parameter can be set in the score!

Note that h is a higher order function too. It takes one argument and produces a function. If our language supports lists (and every FP language does) we can define another function:

$$q(e)([a_0,...,a_n])(w)=a_0(t_i)+a_1(t_i)\cdot e(1wt_i)+a_2(t_i)\cdot e(2wt_i)+...+a_n(t_i)\cdot e(nwt_i)$$

that takes in basis function and produces another function that takes in list of functional coefficients and produces a function.

If you look at patch building as composition of units even from this simple example you can see how this new level of abstraction can greatly enrich the structure of your patches. You can spot a glimpse of hidden symmetries in your sound design, express them with your functions and receive succinctness of your language as byproduct.

Score

In this subsection we'll see how another feature of FP, namely algebraic data types, can make the language of score composition more human.

Algebraic data types consist of two operations for building complex data types out of simple ones. There are predefined types like Integer, Char, Double and two operations: product and sum of types.

Product:

```
Tnew = SomeName T1 T2 … Tn
```

where `T1`, `T2`, …, `Tn` – already defined types. Product of types means that value of type `Tnew` consists of values of types `T1`, `T2`, …, `Tn`. A name `SomeName` is called a constructor. It is a label. It indicates connection between type `Tnew` and subtypes `T1`, `T2`, …, `Tn`.

Sum:

```
Tnew = Name1 T11 | Name2 T2 | … | NameN Tn
```

where `T1`, `T2`, …, `Tn` – already defined types. Sum of types means that value of type `Tnew` can be value of type `T1` or `T2` or … or `Tn`. `NameI` is a label used to distinguish different cases.

Let's look at some examples. Definition of booleans, here we use one sum and primitive types are just labels. They contain no subtypes:

```
Bool = True | False
```

Types can be recursive. Here is a definition of Peano natural numbers:

```
Nat = Zero | Succ Nat
```

Types can be polymorphic, i.e. depend on some another type, like list of values of some type. Here is definition of lists:

```
List a = Null | Cons a (List a)
```

We use one sum and one product in `Cons` case. List of type `a` is empty list (labeled as `Null`) or value of type `a` and another list of type `a` (in Haskell for brevity `Null` is denoted as `[]`, `Cons` as `(:)` and `(List a)` as `[a]`).

But let's come back to music! Following description is the simplified summary of Paul Hudak's ideas on a definition of domain specific language for natural expression of musical ideas [1].

What is a musical notation? Keeping algebraic data types machinery in mind let's try to answer this question. From this point of view we should define set of primitive cases and set of operations that compose things together.

Primitive units in the musical notation are notes. Different musical traditions can introduce different notions of notes, so our type should be polymorphic. It depends on actual note representation. It can be pitches specific to particular countries paired with volumes or it can be frequencies and amplitudes or no frequency at all (drum's case). But there are common things in the note representation. Every note has duration. It lasts for some time. And note can be absent for some time, we call this silence or rest.

So let's define primitive cases:

```
Score t a = Rest t | Note t a | ...
```

where `t` denotes time and `a` denotes note representation.

And what are operations? There are two basic ways of playing several notes: you can play them sequentially or simultaneously. So let's complete definition with two operations:

```
Score t a = Rest t
          | Note t a
          | Seq (Score t a) (Score t a)
          | Par (Score t a) (Score t a)
```

where `Seq` stands for sequential, and `Par` stands for parallel.

In this terms we can define another operations:

```
rest :: t -> Score t a
rest = Rest

note :: t -> a -> Score t a
note = Note

(+:+) :: Score t a -> Score t a -> Score t a
(+:+) = Seq

(=:=) :: Score t a -> Score t a -> Score t a
(=:=) = Par

delay :: t -> Score t a -> Score t a
delay t sco = Seq (Rest t) sco

line  :: Num t => List (Score t a) -> Score t a
line Null = Rest 0
line (Cons x xs) = Seq x (line xs)


chord  :: Num t => List (Score t a) -> Score t a
chord Null = Rest 0
chord (Cons x xs) = Par x (chord xs)
```

Here notation `(a :: T)` means value `a` has type `T`. `Num t =>` means that type `t` is like numbers (supports addition, multiplication, can be constructed from integers), we use this property in `Null` cases (we use `0` symbol for value of type `t`).

With this functions you can write something like this:

```
sNew = stretch 2 (
        chord [ loop 4 (line [s1, s2, s3, ...]),
                line [s1, s2, ...],
                ...])
```

Where `s1`, `s2`, … can be notes and can be another scores and then you can use `sNew` in similar structures! `stretch` is another function that can be defined in terms of two basic operations. It just stretches scores in time domain. `[a, b, …]` is shortcut for `(Cons a (Cons b …))`.

You can think of scores not in terms of events but in terms of sequent and parallel compositions of note chunks and transformations of this chunks defined in terms of this basic operations [2]. Alongside with composable structures for scores we can use naming abstraction. You can give a name to a chunk of notes and freely use it in your functions as any other argument.

III. Overview of csound-expression

What do you need to start playing with csound-expression? Csound-expression(CE) is a Csound code generator written in Haskell. So first of all you need Csound if want to play produced .csd files. Then you need Haskell compiler, so you need GHC [3]. GHC is a state-of-the-art, open source Haskell compiler. You can install it directly or with Haskell Platform [4]. If installation succeeds you can install csound-expression [5] from Hackage by cabal-install.

If you don't know Haskell you can read this great book: “Learn you a Haskell for a great book” [6]. It's available on-line. This beginner's guide is enough to understand what's going on and how to use proposed library.

Here you'll find basic principles of the library, look in documentation for details.

Orchestra

This Section describes how to build instruments with CE. In CE an instrument is a function. It maps a note representation to sound. Sound is described with type `SignalOut`, so instrument has type:

```
instr :: a -> SignalOut
```

where `a` is some note. To play note with instrument means to apply instrument to a note. Seems to be a tautology, but yes it's just as simple as this. There are only two predefined p-fields `itime` and `idur` (`p2` and `p3` in Csound). Other p-fields are derived from instrument structure.

But how to construct this things? Instruments are built by composition of Csound opcodes. There are several types that came from Csound: `Arate` (audio rate signals), `Krate` (control rate signals), `Irate` (constants) and `BoolRate` (for comparisons for if-expressions). There are two special types `MultipleOut` and `SideEffect`. The type `MultipleOut` models Csound opcodes that can return several signals. `SideEffect` models opcodes that can produce random numbers.

Elementary units of instrument composition are Csound opcodes. Csound opcodes are represented with functions. In Haskell functions take in fixed number of arguments so optional initial arguments are represented with lists of Irate's. They are written in the beginning of a function (for easy carrying) other arguments are placed in the same order as arguments in Csound opcodes. To allow Csound polymorphism two type classes are introduced `X` (`Arate | Krate | Irate`) and `K` (`Krate | Irate`). In Csound some opcodes can produce result of different rates and rate is defined by left side of assignment. In CE these functions have suffixes. For example opcode `oscil` has two counterparts in CE `oscilA` and `oscilK`.

Here is a simple example of an instrument:

```
instr :: (Irate, Irate) -> SignalOut
instr (amp, cps) = out (oscilA [] (env <*> amp) cps ft)
      where env = linsegK [0, 0.2*idur, 1, 0.8*idur, 0]
            ft  = gen10 4096 [1]
```

This definition looks very much like Csound definition, but you can see that opcodes can be nested (as in case of `out` opcode). Types of used functions:

```
out     :: Arate -> SignalOut
oscilA  :: (X x0, X x1) => [Irate] -> x0 -> x1 -> Irate -> Arate
linsegK :: [Irate] -> Krate
gen10   :: Irate -> [Irate] -> Irate
idur    :: Irate
```

Operator `<*>` - is a special variant of multiplication defined on signals. We use it here because `env` and `amp` have different types and standard multiplication in Haskell takes values of the same type.

So we get Csound opcodes as units and can compose them in all fancy ways that Haskell allows us. Let's define functions *q*, *f* and *h* described in Section I.

```
q :: (Irate -> Arate) -> [Arate] -> Irate -> Arate
q e as w = sum (zipWith combine as [0,1..])
     where combine a n = a * e (fromInteger n * w)
```

```
h :: Irate -> Irate -> Arate
h a w = oscilA [a] (num 1) w (gen10 4096 [1])

f :: Irate -> [Arate] -> Irate -> Arate
f a = q (h a)
```

From type of `q` we can see that `q` takes in function as first argument (basis function), list of signals as second argument (coefficients) and produces function, that builds series on given basic frequency. So we can define generator of arbitrary functional series. You can see from this example how flexible functions can be.

Embedding Csound in functional environment enhances features natural to functional languages. But impedes some features that Csound inherited from imperative world. There are some opcodes in Csound that rely on order of execution (like delayr/delayw) and they can not be expressed so easily in functional environment. For this cases CE allows user to write almost literal Csound code, but this code is no longer composable. It becomes a sequence of lines containing opcode assignments.

Score

Instruments are just functions from note representation to sound. But what is score? Well score is a container of notes. To play score on some instrument means to transform container of notes to container of sounds.

Structure of container is roughly described in Section II. CE relies on another Haskell library temporal-media [7]. This library provides functions to construct and transform musical structures and flatten them to list of events (just what Csound needs). CE transforms container of sounds to list of events (each event contains instrument description with specific parameters) and then derives list of instruments and ftables, substitutes events with notes and packs everything in csd file.

Let's look at an example of simple program:

```
-- D minor chord
import Temporal.Media

import CsoundExpr.Base
import CsoundExpr.Base.Pitch
import qualified CsoundExpr.Opcodes as C

-- | sinusoidal oscillator
instr :: Irate -> SignalOut
instr cps = C.out $ env <*> C.oscilA [] (num 3000) cps (gen10 4096 [1])
      where env = linsegK [0, idur/3, 1, 2*idur/3, 0]

-- | d minor chord
sco = fmap instr $ sequent notes +:+ parallel notes
    where notes = map (temp 1) [d, f, a, high d]

file = "out.csd"
flags = "-d"

out = csd flags headerMono $ renderMedia sco

main = writeFile file $ show out
```

Function `csd` produces csd file, `renderMedia` flattens container of sounds to list of events, `fmap`

applies a function to all elements of some container, it's key element in this notation. Function `temp` is synonym to `note` defined in Section II, as `sequent` is to `line` and `parallel` to `chord`.

Note that there is no separate sections of score an orchestra. Instrument is not a separated entity. It is a function that interprets note (i.e. maps note representation to sound).

IV. Conclusions, related and future work

This paper describes how natural features of functional programming languages, namely higher order functions and algebraic data types, can greatly enhance expressive power of Csound. Higher order functions give a powerful way to abstract various patterns of the instrument structure. Algorithms that are parametrized by particular behavior may be easily defined with higher order functions. Algebraic data types make it possible to write music in terms of pieces of music and their compositions and transformations, without using the terms of single events.

An implementation of this ideas can be found in csound-expression, Haskell library and Csound code generator. There are other libraries that can produce Csound code from specification written in Haskell. Haskore [8] and Euterpea [9] contain sections related to Csound. In them there is only one type for all rates. In CE there are different types for different rates. It excludes production of some invalid Csound code, that can be made by mistake. CE has more advanced system of score/orchestra interaction. Instruments in Haskore and Euterpe has interface similar to Csound (note calls instrument with p-fields), but in CE actual note representation is not fixed it can be anything that can be interpreted with your instruments.

CE is still evolving project. And some features of Csound language are missing. It doesn't support if/then/else when your expressions are not composable (i.e. when it's not a composition of simple units, but sequence of opcode assignments). I'm planning to make musical examples in CE, include imperative if/then/else, make code optimization, and investigate a possibility of composable Csound GUI building.

References

[1] Paul Hudak, An Algebraic theory of polymorphic temporal media, Research Report RR-1259, 2003.

[2] Laurie Spiegel, Manipulations of musical patterns, Proceedings of the Symposium on Small Computers and the Arts, 1981.

[3] Glasgow Haskell Compiler, http://www.haskell.org/ghc/.

[4] Haskell Platform, http://hackage.haskell.org/platform/.

[5] csound-expression, Hakell library, http://hackage.haskell.org/package/csound-expression.

[6] Miran Lipovaca, Learn you a Haskell for a great good, http://learnyouahaskell.com/.

[7] temporal-media, Haskell library, http://hackage.haskell.org/package/temporal-media.

[8] Haskore, Haskell library, http://hackage.haskell.org/package/haskore.

[9] Euterpea, Haskell library, http://haskell.cs.yale.edu/?page_id=103.

V. Appendix

Generated Csound code for a “d-minor” example

```
<CsoundSynthesizer>
<CsOptions>
-d
</CsOptions>
<CsInstruments>
; ------------------------------
; Header
sr       =   44100.0
kr       =   4410.0
ksmps    =   10.0
nchnls   =   1.0


; ------------------------------
; Instruments

      instr    33
a3    oscil    3000.0, p4, 1
k12   linseg   0.0, (p3/3.0), 1.0, ((2.0*p3)/3.0), 0.0
      out      (k12*a3)
      endin



</CsInstruments>
<CsScore>
; ------------------------------
; Ftables
f 0 5.0
f 1 0.0 4096 10 1.0


; ------------------------------
; Tempo



; ------------------------------
; Notes
i 33 0.0 1.0 8.02
i 33 1.0 1.0 8.05
i 33 2.0 1.0 8.09
i 33 3.0 1.0 9.02
i 33 4.0 1.0 8.02
i 33 4.0 1.0 8.05
i 33 4.0 1.0 8.09
i 33 4.0 1.0 9.02

e


</CsScore>
</CsoundSynthesizer>
```