Print

Dear list I'd like to announce the new release of my library that embedds Csound into Haskell. This is quite a big release. Below is complete list of features

The 5.2 is out. Virtual pedalboards, arrays, new OSC, full support for mono synthesizers, patch skins, all GEN-routines are implemented

New features:

- **Complete support for monophonic synthesizers**:
  - The argument of mono synth was updated.

    Previously it was a pair of amplitude and frequency signals.
    But this representation doesn't captures the notion of note retrigger.
    We can not create the mono-synt with sharp attacks.

    Now this is fixed. We can use adsr140 or adsrMonoSynt functions to create
    mono synths with fixed attacks
  - monoSco - for playing scores with mono-synths
  - monoSched - for playing event streams with mono synt
  - atSco and atSched now work for mono synth too
- **The patch can change the skin**. The Patch type has changed. Know it supports the change in common parameters.
  Right now the ccommon parameters include only Low-pass filter type. But this can be extended in future releases.

  The idea is that we can parametrize the patch with some common arguments so that use can tweak them
  without revriting the algorithm.

  The low-pass filter is a vital tool that defines the character of the synthesizer.
  With recent addition of several modern filter emulators (like Korg (korg_lp), or acid filter diode)
  it's good to be able to quickly switch the filters. We can do it for patches with function

```
setFilter :: ResonFilter -> Patch a -> Patch a
```

- **Family of standard effects was added** (see module `Csound.Air.Fx.FxBox` and the guide).
  The effects are kindly provided by Iain McCurdy (recoded from his original implementation in Csound).

  The effects have catchy names and are defined on wide variety of types.
  Let's briefly discuss the naming conventions:

- `adele` - analog delay
- `pongy` - ping pong delay
- `tort` - distortion
- `flan` - flanger
- `fowler` - Envelope follower
- `phasy` - phaser
- `crusher` - bit-crusher
- `chory` - stereo chorus
- `tremy` - tremolo
- `pany` - panning
- `revsy` - reverse playback

Also there are set of presets that imply the notion of add a bit of effect or add a lot of effect.
They are suffixed with number from 1 to 5. Like `flan1` or `tort3`. Also if the effect support the
tone knob (center frequency of LP filter) ter are suffixes `b` for bright color and `m` for muted color.
For example `tort2m` or `adele2b`.

The effects are just functions from signals to signals:

```
dac $ hall 0.2 $ adele2 0.5 0.25 $ flan2 $ tort1m $ asigs
```

- **UI widgets for standard effects**.

  Alongside with effects there are functions to create widgets (UI-controls). They have the same naming convention
  only the prefix `ui` is added. For example: `uiTort`, `uiAdele` or `uiHall`.
  Also there are predefined presets like `uiFlan2` or `uiPhasy3`.
  With presets we put the box in the initial state corresponding to the given preset. But lately we can change it with UI-controls.

  With this feature paired with functions `fxHor`, `fxVer` and `fxGrid` we can easily design our virtual pedalboards.

  It can be used like this:

```
> let pedals = fxGrid 2 [uiFlan1, uiTort1, uiAdele2m 0.5 0.3, uiHall 0.

> dac $ fxApply pedals $ (sawSeq [1, 0.5, 0.25] 2) * sqr 220
```

- **Complete list of GEN routines**. This release adds GEN:

```
 * 25 bpExps --  Construct functions from segments of exponential curve
```

```
* 27 bpLins --  Construct functions from segments of straight lines in

* wave waveletTab -- Generates a compactly supported wavelet function.

* farey fareyTab -- Fills a table with the Farey Sequence Fn of the in

* sone soneTab -- Generate a table with values of the sone function.

* exp expTab -- rescaleExpTab Generate a table with values on the exp

* tanh tanhTab -- rescaleTanhTab Generate a table with values on the t

* 52 readMultichannel -- Creates an interleaved multichannel table fro

* 41 randDist -- Generates a random list of numerical pairs.

* 42 rangeDist Generates a random distribution of discrete ranges of v

* 40 tabDist -- Generates a random distribution using a distribution h

* 43 readPvocex -- Loads a PVOCEX file containing a PV analysis.

* 28 readTrajectoryFile -- Reads a text file which contains a time-tag

* 24 readNumTab --  Reads numeric values from another allocated functi

* 21 dist, uniDist, linDist, triDist, expDist, biexpDist, gaussDist, c

* 18 tabseg -- Writes composite waveforms made up of pre-existing wave

* 31 mixOnTab -- Mixes any waveform specified in an existing table.

* 32 mixTabs -- Mixes any waveform, resampled with either FFT or linea

* 30 tabHarmonics -- Generates harmonic partials by analyzing an exist
```

See the Csound docs for details of what table they produce.
Also the signatures for windows creating tabs was updated. It became more specific.

- **Global arguments** defined with **Macros**. We can create a Csound `.csd` file in our program
and after that we can run it on anything which has Csound installed. It's desirable to be able
to tweak some parameters after rendering or to have some global config arguments.
In Csound we can do it with macroses. We can use macros name in the code adn then we can change the value of the
macros with command line flag `--omacro:Name=Value`.

From now on it's possible to do it with Haskell too. There are functions:

```
readMacrosDouble  :: String -> Double -> D
readMacrosInt     :: String -> Int -> D
readMacrosString  :: String -> String -> Str
```

The first argument is a macro name and the second one is the default value
which is used if no value is set in the flags.

- The useful function to **trigger an table based envelope**. It comes in two flavors. One is driven with event stream
and another with just a signal. It's on when signal is non zero.

```
trigTab :: Tab -> Sig -> Sig -> Sig
trigTab tab duration triggerSignal

type Tick = Evt Unit

trigTabEvt :: Tab -> Sig -> Tick -> Sig
trigTabEvt tab duration triggerSignal
```

- **New functions for UI widgets**.
  - We can change the relative size of the window. If the widget is too large or too small
    we can rescale it with functions:

```
type ScaleFactor = (Double, Double)

resizeGui :: ScaleFactor -> Gui -> Gui

resizeSource :: ScaleFactor -> Source a -> Source a
```

    They change the default minimal sizes for all widgets that are contained within the given widget.
  - Grid layout. We are familiar with functions `ver` and `hor`. With them we can place the widgets vertically or horizontally.
    But now it's also possible to place the widgets on the grid:

```
grid :: Int -> [Gui] -> Gui
```

    The first argument specifies the number of elements in each row.

    There are handy grid functions for combining source-widgets:

```
gridLifts :: Int -> ([a] -> b) -> [Source a] -> Source b
```

It applies a list based function to a list of value producer widgets and places all widgets on the grid.
The first argument is the same as with `grid`.

- UI default sizes are a bit smaller now.

- It compiles on **GHC-7.8** again
- New function `whileRef` for imperative while loops.

```
whileRef :: Tuple st => st -> (st -> SE BoolSig) -> (st -> SE st) -> SE
whileRef initState condition body
```

It's used in this way. It stores the initial state in the reference (local variable)
and the it starts to implement the body while the predicate returns true. Notice that
the body is also updates the state.

- **New functions for OSC** that make it easy to read OSC messages that are interpreted like signals.
For example we have an OSC-routine for volume control. When message happens we update the value.
It would be good to be able to just read the signal:

```
listenOscVal :: (Tuple a, OscVal a) => OscRef -> String -> a -> SE a
listenOscVal oscRef address initValue
```

There are two useful aliases for this function. They read signals and pairs of signals:

```
listenOscSig  :: OscRef -> String -> Sig  -> SE Sig
listenOscSig2 :: OscRef -> String -> Sig2 -> SE Sig2
```

- **Adds loopers that preserve attacks when rescaling by tempo**.
They are based on `temposcal` Csound algorithm.
The previous loopers were based on the `mincer` algorithm. It uses FFT under the hood which can smooth out the sharp attacks.
It's undesirable for percussive loops. The `temposcal` adds the feature of preserving attacks.

  See the functions:

```
-- | reads stereo files with scaling
scaleWav ::  Fidelity -> TempoSig -> PitchSig -> String -> Sig2

-- | reads mono files with scaling
scaleWav1 :: Fidelity -> TempoSig -> PitchSig -> String -> Sig
```

Also there are presets for scaling the drums or harmonic instruments (they set the appropriate fidelity):

```
scaleDrum, scaleHarm :: TempoSig -> PitchSig -> String -> Sig2
```

The fidelity is the degree of the size of FFT window. The formula for window size: 2 ** (fidelity + 11).

Also the corresponding functions are added for `csound-sampler`.

```
wavScale :: Fidelity -> TempoSig -> PitchSig -> String -> Sam
wavScale1 :: Fidelity -> TempoSig -> PitchSig -> String -> Sam

drumScale, harmScale :: TempoSig -> PitchSig -> String -> Sam
```

- The type signatures for **echo and pingPong where simplified**. Now they don't use side effects
  and look like pure functions:

```
echo :: MaxDelayTime -> Feedback -> Sig -> Sig
pingPong :: DelayTime -> Feedback -> Balance -> Sig2 -> Sig2
```

- Type signatures for functions `randSkip` **and** `freqOf` **where generalized**. Now they use signals for probabilities
  instead of constant numbers. So we can change the probability of skip of the event during performance.
- **New monophonic instruments are added in csound-catalog**:
  `fmBass1`, `fmBass2`, `dafunkLead` and one polyphonic `celloSynt`.
  Those instrument serve a good example for building monophonic synthesizers with sharp attacks.

Experimental features:

- **Arrays, with all opcodes** and functional traversals. See the guide for details details.
- **Imperative style instruments**.

  With imperative style instruments we can create and invoke the instruments in Csound way.
  we can create an instrument and get it's unique identifier. Than we can schedule a note by that identifier.

  We can create an instrument that produces a sound with function:

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

  It takes in a body of the instrument and gives back an instrument

reference and
a signal where the output is going to be written. Then we can invoke the notes just
like we do it in the Csound with function `scheduleEvent`:

```
scheduleEvent :: Arg a => InstrRef a -> D -> D -> a -> SE ()
scheduleEvent instrRed delayStartTime duration arguments
```

It takes in instrument reference, start time from the time of invocation, duration (all in seconds) and miscellaneous arguments.
Notice that the instrument reference is parametrized by the type of arguments. This way we can not feed the wrong messages to the instrument.

Also we can create procedures that produce no output but do something useful:

```
newInstr :: Arg a => (a -> SE ()) -> SE (InstrRef a)
```

Happy Csounding!
Anton

Links to the library: https://hackage.haskell.org/package/csound-expression

See the guide at github: https://github.com/spell-music/csound-expression

Csound mailing list [log in to unmask] https://listserv.heanet.ie/cgi-bin/wa?A0=CSOUND Send bugs reports to https://github.com/csound/csound/issues Discussions of bugs and features can be posted here