LIVE CODING AND CSOUND

Hlöðver Sigurðsson
hlolli@gmail.com

In this paper I'm going to cover a way to design a live-coding front end for Csound. My programming language of choice is Clojure, which I've used to develop Panaeolus, a live-coding program built with CsoundAPI. The aim of Panaeolus is not only to bring Csound into the world of functional programming and lisp, but also to build an extendable musical system that can create complex musical pattern, with as short and descriptive code as possible.

The origin of Panaeolus dates back to April 2015 when I was using Overtone (SupercolliderAPI for Clojure) for live-coding. Initially I just added few Csound instruments into my live-coding sets, but as my preference for the acoustical qualities of Csound are greater than those of Supercollider, I decided to leave the world of Supercollider and began to develop my own live-coding environment in July that same year. At the time of this writing, Panaeolus still needs better documentation, testing and stable release. It can be found under GNU license on http://github.com/hlolli/panaeolus.

Even tough I will explain concepts in this paper that apply to Clojure, I will to point out that almost identical principles apply to other programming languages, even the Csound language itself. And at the time of this writing, a short article of live-coding in the Csound language with CsoundQt front-end is scheduled for Csound Journal spring issue of 2016.

1 Performance loop

Since Clojure is built on top Java, the interoperability of Java's method and classes from csnd6.jar become automatically available in Clojure. Tough sometimes, depending on the operating system, the location of csnd6.jar needs to be available to Java's environment variables. For minimal setup of ClojureAPI it's possible to load all the basic classes and methods from csnd6.jar into the namespace with the following import statement.

```
(ns panaeolus.metro ;in panaeolus this will be in a file called metro.clj
   (import [csnd6 csnd6 Csound])) ;import CsoundAPI into the namespace
```

If successful then it's possible to initialize, create and start an instance of Csound.

```
(csnd6/csoundInitialize
   (bit-or csnd6/CSOUNDINIT_NO_ATEXIT
            csnd6/CSOUNDINIT_NO_SIGNAL_HANDLER)) ; initialize Csound
(def c (Csound.)) ; create an instance of Csound and assign it to 'c'
(.Start c) ; start an instance of Csound
```

After evaluation, the (n)REPL should print something like the following.

```
user=> virtual_keyboard real time MIDI plugin for Csound
0dBFS level = 32768.0
Csound version 6.07 (double samples) Feb 3 2016
libsndfile-1.0.26
Reading options from $HOME/.csound6rc
rtaudio: JACK module enabled
rtmidi: ALSA Raw MIDI module enabled
sample rate overrides: esr = 44100.0000, ekr = 4410.0000, ksmps = 10
--Csound version 6.07 (double samples) Feb 3 2016
graphics suppressed, ascii substituted
0dBFS level = 32768.0
orch now loaded
audio buffered in 2048 sample-frame blocks
system sr: 44100.000000
0: dac:system:playback_ (system:playback_)
writing 2048 sample blks of 64-bit floats to dac:system:playback_
SECTION 1:
```

This indicates that Csound is now running (without performance thread) trough the JVM and it's now possible to send instrument definitions and in form of strings to this Csound instance (without performance thread, they will not sound). Software buses, control channels and MYFLT messages are also possible trough the API, if imported into the namespace, but is outside of the scope of this paper.

The API callback `(.PerformKsmps csound-instance)` will make a performance pass and returns 0 on each successful cycle. So by creating an infinite loop on a thread it's possible to drive performance passes and mutate a counter that can be used control event scheduler. This ensures that everything that Csound is doing will be on the same thread and therefore minimizing any asynchronicity.

```
(defn perf-loop [csound]     ;Define a function that takes an instance of csound as an argument
  (fn []               ;This will be a function that returns a function ie. Closure
     (loop [last-v 0]        ;Create a recursive loop with control-rate counter initialized to 0
         (if (zero? (.PerformKsmps csound))    ;.PerformKsmps returns 0 on every pass
           (let [new-v (inc last-v)]           ;Assign new-v to be a mutated value a last-v
              (prn new-v)                      ;For debugging, let's print the value of new-v
              (recur new-v))))))               ;Run the loop again with last-v becoming new-v
```

The function `perf-loop` is a closure so a callback is needed for filling in the missing arguments, in this case the Csound instance ([csound]) `c`. Adding callbacks into the `perf-loop` enables all events handling to live on one thread in a functional manner. The state of the recursion (new-v) does not have to be stored in global variable, but rather in pure functions which is good in terms of performance.

```
(def perf-thread                ;assign perf-loop to perf-thred
   (Thread. (perf-loop c)))     ;callback instance of csound 'c' in a thread

(.start perf-thread)            ;start the thread
```

After starting the performance thread, a stream of number should be printing in the nREPL, indicating that Csound is running and ready to make some noise. To kill the thread simply evaluate `(.stop perf-thread)`.

2 Track-buffer model

Panaeolus is based on what I call tracks, there is to say each pattern is assigned to a unique track. What all tracks have in common is that they all read the same state from the same performance thread, meaning event schedule for the same time will in fact be played simultaneously. Nevertheless, various variables offered by the clojure.core (ref, atom, agent and var) offers various kinds of (a)synchronicity. For a short answer on what separates one variable from another, I quote the following stackoverflow comment [1]:

"*Refs are for* ***Coordinated Synchronous*** *access to "Many Identities".*

Atoms are for ***Uncoordinated synchronous*** *access to a single Identity.*

Agents are for ***Uncoordinated asynchronous*** *access to a single Identity.*

Vars are for thread local ***isolated identities*** *with a shared default value.*

Coordinated *access is used when two Identities need to be changes together, the classic example being moving money from one bank account to another, it needs to either move completely or not at all.*

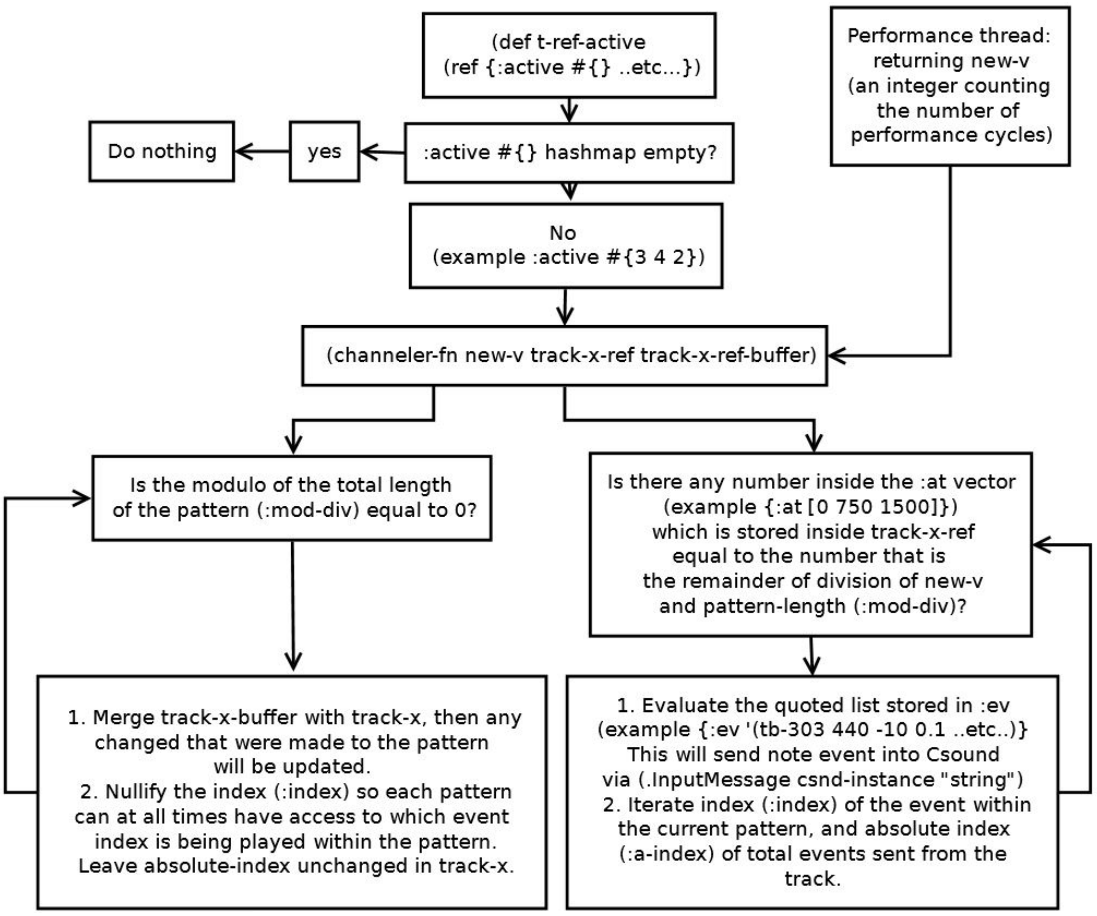
Uncoordinated *access is used when only one Identity needs to update, this is a very common case.*

Synchronous *access is where the call expects to wait until all the identities are settled before continuing.*

Asynchronous *access is "fire and forget" and let the Identity reach its new state in its own time."*

While designing Panaeolus using refs was an important trade-off for better stability. The high performance offered by the other variable types available in Clojure did not seem to suit live-coding, but further research and testing may be needed.

To have the possibility of creating musical patterns in real-time, thinking in patterns can be very helpful. The problem that live-coders may encounter is when evaluation of code is so immediate that new musical idea has overwritten the old musical idea before it has played trough. To solve this problem I came up with what I call track-buffer model. The idea is that for a pattern of any given length will not become apparent until the old pattern has come to an end. Therefore some sort of buffer needs to exist, to store the changes until the right moment arrives. So for each track there are two refs variables that are assigned to almost identical hash-maps. The one being read is always track-x-ref (where x is a number) and the other one will be storing any changes, track-x-ref-buffer.


(def t-ref-active
(ref {:active #{} ..etc...})
Performance thread: returning new-v (an integer counting the number of performance cycles)
Do nothing
yes
:active #{} hashmap empty?
No
(example :active #{3 4 2})
(channeler-fn new-v track-x-ref track-x-ref-buffer)
Is the modulo of the total length of the pattern (:mod-div) equal to 0?
Is there any number inside the :at vector (example {:at [0 750 1500]}) which is stored inside track-x-ref equal to the number that is the remainder of division of new-v and pattern-length (:mod-div)?
1. Merge track-x-buffer with track-x, then any changed that were made to the pattern will be updated.
2. Nullify the index (:index) so each pattern can at all times have access to which event index is being played within the pattern. Leave absolute-index unchanged in track-x.
1. Evaluate the quoted list stored in :ev (example {:ev '(tb-303 440 -10 0.1 ..etc..)} This will send note event into Csound via (.InputMessage csnd-instance "string")
2. Iterate index (:index) of the event within the current pattern, and absolute index (:a-index) of total events sent from the track.

Flowchart 1 The track-buffer model like it appears in Panaeolus

In Clojure a simplified version of `channeler-fn` would look like this:

```
 (defn channeler-fn [new-v t-ref t-ref-buffer] ; A function that takes 3 arguments
    (dosync                                     ;dosync is a macro that is needed to mutate refs.
      (if (zero? (mod new-v (:mod-div @t-ref))) ;if modulo of the total length of the pattern is 0…
          (alter t-ref merge @t-ref-buffer))    ;…then merge everything from track-buffer into track
       (if (some #(= (mod new-v (:mod-div @t-ref)) %) (:at @t-ref)) ;if any number from vector :at
         (do                                                ;is equal to (new-v%pattern length)
            (eval (:ev @t-ref))                             ;then evaluate the next note event
            (alter t-ref (fn [x-map] (assoc x-map :index (inc (:index x-map)) ;after that iterate (inc)
                                           :a-index (inc (:a-index x-map)) ;the index and a-index
                                                    ;but also read any instrument changes
                                           :ev (:ev @t-ref-buffer))))))))
```

As seen in `channeler-fn` in the last line. Any changes made to the score event (:ev) are mutated for every event. This is a design choice made to make live sound-design faster. In a long pattern, changing parameters like cutoff filter, detune, reverb ...etc..., in a fast responsive way can be very effective. Now by editing the `perf-loop`, it's possible to send some dummy patterns into `channeler-fn`.

```
(.CompileOrc c " ;Some dummy Csound instrument
                       ; sent to a csound instance via string.
instr synth
iamp = ampdb(p4)
ifreq = cpsmidinn(p5)
asig vco2 iamp, ifreq
aenv expon 1, p3, 0.1
afilt moogvcf2 asig, 310, 0.9
outs afilt*aenv, afilt*aenv
endin")

;A track
(def track1 (ref {:mod-div 3000 :at [] :index 0 :a-index 0 :ev nil}))
;A track-buffer
(def track1-buffer (ref {:mod-div 3000 :at [0 750 2250]
                                                 :ev `(.InputMessage c "i \"synth\" 0 1 -120 50")
                                                 :index 0}))
(defn perf-loop [csound]
   (fn []
      (loop [last-v 0]
         (if (zero? (.PerformKsmps csound))
            (let [new-v (inc last-v)]          ;Here is new-v fed into the channeler-fn
               (channeler-fn new-v track1 track1-buffer)
               (recur new-v))))))

(def perf-thread                          ;in case the thread was killed
   (Thread. (perf-loop c)))

(.start perf-thread)                           ;start the thread again
```

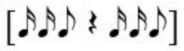
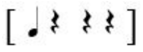
Here a pattern of length 3000 is scheduling events on 0, 750 and 2250. These numbers and how to design a scheduler will be the topic of next section. But the important thing here is that these number in `:at` vector can be redefined multiple times, but any changes to this vector will only be apparent when the modulo of `new-v` and 3000 is equal to 0. But any changes to `:ev` will immediately become apparent (on next event).

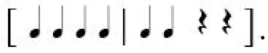
3 Designing a scheduler

The rate of which `.PerformKsmps` is looping is determined by the value of `kr`. So the number returned by new-v will always be equal to the number returned by the Csound opcode `timek`. Irrelevant of if the value of `kr` (or `sr` divided by `ksmps`) is stored in *.csound6rc* textfile or assigned to the orchestra via `.CompileOrc`, the value of `kr` has to be available for accurate scheduling algorithms.

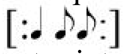
With this number it is possible to create a function that calculates how many performance passes (I call 'ticks') are to each beat.

```
(def bpm-atom                          ;the global bpm is stored in atom variable
   (atom {:tps 750 :bps 2.0}))        ;initialized to 750 ticks per beat
                                                ;for my kr-rate 750 means
                                                ;2 beats per second(120BPM), this may vary.
(def CSOUND-SR 44100)    ;An example of a way to store the sample rate value
(def CSOUND-KSMPS 32)    ;An example of a way to store the ksmps value.
(defn bpm! [bpm]         ;Make function 'bpm!' that can mutate…
                                ;…the tempo in performance
   (let [sr CSOUND-SR
         ksmps CSOUND-KSMPS
         sec (/ sr ksmps)       ;This basically calculates the value of kr
         bps (/ bpm 60)               ;Calculating beats per second
                                        ;Divide kr to bps and …
                                        ;…use ceil to round integer up.
         tps (Math/ceil (/ sec bps))]
      ;Mutate the state of bpm-atom.
      (swap! bpm-atom assoc :tps tps :bps (float bps))))
```

Next problem for designing live-coding application is what kind of algorithmic patterns are best suited for musicians. One common way is to stack note duration on one another. So a pattern of [0.25, 0.25, 0.5] would mean something like . I argue that this way of stacking note duration on one another, becomes very unclear in longer patterns. For example after series of [0.25 0.25 … etc…] it becomes hard to know if next note event is going to be on beat or if it's going to match to other patterns. More clear way would be to use what I call 'on-s' in Panaeolus. So a pattern of [0, 0.25, 0.5, 1, 1.25, 1.5] would be something like . Which begs the next question; for looping patterns, how can one add rest to the end of a pattern? For example something like . To solve this in Panaeolus I designed a measured pattern system. By default, every pattern created will be in a musical meter of 4/4. That means when [0] pattern is created in 4/4 the result will be something like  where the gaps are filled with rest to make up a 4 beat bar. Also if the pattern extends one bar, then a second bar will be generated in 4/4.

So [0, 1, 2, 3, 4, 5] pattern will result in . The following function is a simplified version of 'on-s' to 'at' converter as ıt appears in Panaeolus.

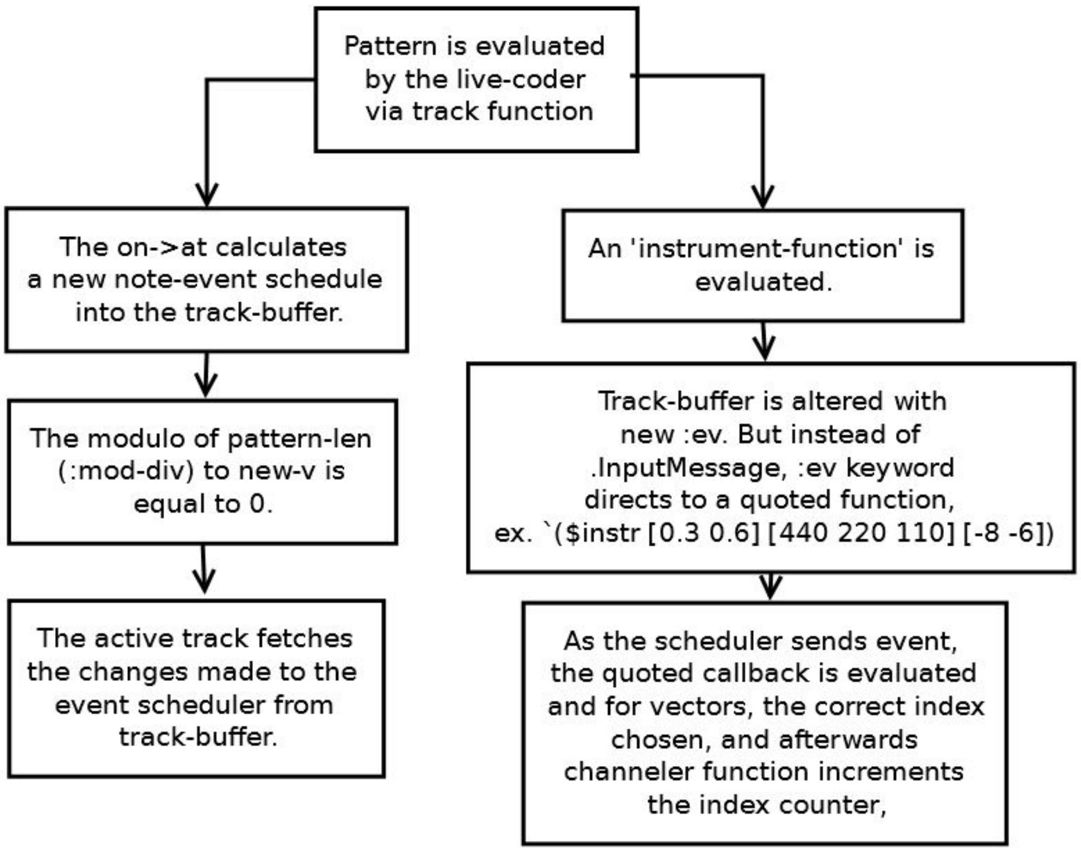
```
(defn on->at [on-s meter t-ref-buffer]    ;convert 'on-s' to 'at-s'
   ;calculate from global bpm how many ticks one bar takes
   (let [meter (* meter (:tps @bpm-atom))
         ;to prevent minimize chances of crash, the on-s can be a
         ;number as well as a vector.
         on-s (if (vector? On-s)
                on-s
                   (if (number? on-s)
                      [on-s]
                      (vec on-s)))
         at (if (= [] on-s)   ;for empty vector, just return an empty hash-map
            #{}
            ;hash-set can be used instead of vector and performs better
            (apply hash-set
                  ;Multiply to each 'on-s' number the ticks per second
                  (map #(Math/round %)
                       ;but make sure that's it's integer, since new-v is always so.
                       (map #(double %)
                             (map #(* (:tps @bpm-atom) %) on-s)))))
         pat-end (if (empty? on-s)
                  ;make sure that max-div is never 0 despite empty patterns
                  (if (zero? meter)
                       ;since channeler-fn needs to read the new state of …
                       ;non-active patterns in case they are restarted.
                       (* 4 (:tps @bpm-atom)) meter)
                  ;when meter is set to 0, then last note event will
                  (if (zero? meter)
                       ;determine pattern-len
                       (* (:tps @bpm-atom) (Math/ceil (+ 0.0001 (apply max on-s))))
                       (* meter (Math/ceil (/ (+ 0.0001 (apply max at)) meter)))))
         mod-div (int pat-end)]
(dosync (alter t-ref-buffer (fn [x-map] (assoc x-map :at at :mod-div mod-div))))))
```

This algorithm allows for meter of 0. In those cases, the total length of the pattern (:mod-div) is determined by the last note event. For example with measure equal to 0 and (looping) pattern [0, 1, 1.5], then the result would be something like  . To achieve this, the pattern length is measure by Math/ceil (round to next greater integer), and adding 0.0001 prevents an integer to be ceiled to itself (e.g. the number 4 being ceiled to 4 instead of 5). The function `on->at` returns a dosync transfer that mutates a track buffer. The following code would send pattern to the dummy instrument.

```
(def track1 (ref {:mod-div 3000 :at [] :index 0 :a-index 0 :ev nil})) ;track
(def track1-buffer (ref {:mod-div 3000 :at [] ;track-buffer
:ev `(.InputMessage c "i \"synth\" 0 0.1 -12 50")
:index 0}))
(bpm! 92) ;try changing the global bpm
(on->at [0 1 2 3 3.5] 4 track1-buffer) ;evaluate this pattern in 4/4 meter
```

4 Patternizing the parameters

To live-code any musical pattern that vary in pitch, timbre or duration, a way to create patterns for Csound instrument p-fields is needed. In the example above, a fixed string is being sent on every note-event. A better way is to have :ev evaluate a function, which then sends input message to Csound instance. With this extra abstraction we can make a function that iterates trough a vector based on index. Another abstraction for a separation between an evaluation of scheduler and instrument is logical, since the aim is to minimize typing cost and ensure that only instrument is evaluated on events but not the schedule vector. In Panaeolus each track is its own function where schedule, instrument, pitch and parameters are all evaluated from one (track) function.


Pattern is evaluated by the live-coder via track function
The on->at calculates a new note-event schedule into the track-buffer.
The modulo of pattern-len (:mod-div) to new-v is equal to 0.
The active track fetches the changes made to the event scheduler from track-buffer.
An 'instrument-function' is evaluated.
Track-buffer is altered with new :ev. But instead of .InputMessage, :ev keyword directs to a quoted function, ex. `($instr [0.3 0.6] [440 220 110] [-8 -6])
As the scheduler sends event, the quoted callback is evaluated and for vectors, the correct index chosen, and afterwards channeler function increments the index counter,

Flowchart 2 Shows the process of pattern being evaluated trough 'track' function.

The following function called `event-shooter` can make instrument design in Clojure for Csound instruments bit more clear.

```
(defn event-shooter ;as featured in Panaeolus
[i & {:keys [p3 p4 p5 p6 p7 p8 p9 p10 p11 p12 p13 p14 p15 p16 p17 p18 p19 p20]}]
(.InputMessage c
;which are concatenated into one event
(str "i " i " " "0" " " p3 " " p4 " " p5 " " p6 " " p7 " " ;Numbers are converted to strings
p8 " " p9 " " p10 " " p11 " " p12 " " p13 " " p14 " " p15 " " p16 " " p17 " " p18 " " p19 " " p20)))
```

Then a function designed to send note events to the dummy instrument from earlier.

```
(defn dummy-synth [& {:keys [dur amp note t-ref]}]    ;destructuring of arguments gives
   (let [dur (if (vector? Dur) ;possibility of named arguments
               (nth dur (mod (:a-index @t-ref) (count dur))) ;iterate trough vector via index
               dur)
         amp (if (vector? amp)
               (nth amp (mod (:a-index @t-ref) (count amp)))
               amp)
         note (if (vector? note)
                  (nth note (mod (:index @t-ref) (count note))) ;for note it's better to use
                  note)] ;pattern index instead of absolute
   (event-shooter "\"synth\"" :p3 dur :p4 amp :p5 note))) ;index
```

And last the track function to update instrument and scheduler in one callback.

```
;In panaeolus this function would be called $1
(defn pattern1 [on-s meter instrument]
   (let [meter (if (nil? meter)
                        4 meter)]
        (on->at on-s meter track1-buffer)
        (dosync (alter track1-buffer (fn [x-map] (assoc x-map :ev instrument))))))
```

Now pattern 1 is ready to create patterns.

```
; Name Schedule Meter Instr-name Dur (p3)           Ampdb(p4)   Cpsmidinn(p5) track1 for index
(pattern1 [0 1 2 3] 4 `(dummy-synth :dur [0.5 0.5 0.8] :amp -10 :note [60 64 67 72] :t-ref track1))
```

Important thing here is the back-tick sign (`) in front of the instrument inside pattern1 callback. In Clojure this is the syntax-quote which prevents the function from being evaluated on evaluation of its caller. Since pattern1 is defined as a function it will get evaluated along with its arguments, unlike macros that have the possibility read-only on evaluation. An unnecessary typing here is the :t-ref parameter since dummy-synth should and could just get this information from its caller (this case pattern1) since each pattern enumeration is always reference to matching track enumeration (this case track1). Macros for better or worse come here to the rescue, compared to Panaeolus an identical callback would look like this.

```
;Panaeolus style
($1 (dummy-synth :on [0 1 2 3] :dur [0.5 0.5 0.8] :amp -10 :note [60
64 67 72]))
```

No syntax quote as well as no `:t-ref` are needed. How this is solved in Panaeolus is outside the scope of the paper. With this design relatively little typing is required and by adding more tracks the live-coder can start to create many independent layers of patterns with various Csound instruments.

5 Musical notation

By introducing musical notation it's possible to shorten the amount of typing even more (at least clarify it musically). In Panaeolus a choice can be made between using the 'on-s'/note pairing and a musical notation. This musical notation I developed with great inspiration from Lilypond (music notation programming language) and Schönberg harmony analysis, and must be added here that it's still being improved. The musical notation comes as string and is 'under the hood' translated into 'on-s'/note. The pattern above could be shortened down into this.

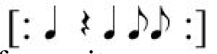
```
;Panaeolus style with music notation
($1 (dummy-synth :pat "s i iii v i'" :dur [0.5 0.5 0.8] :amp -10))
```

To break this string down a bit, it starts with the letter 's' meaning it's acting in scale-mode. So each roman numeric letter will symbolize a position within a given scale, which by default is c-major. The last roman letter i' a comma is added to indicate that the note should be played octave above. Vice versa an octave lower would be a dot, i. . So being hidden behind all the defaults the full expression of this pattern would be.

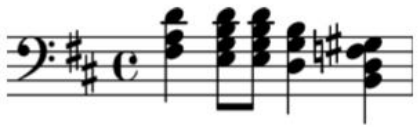
```
"s root=c4 scale=major m4 4 i iii v i' "
```

Here c4 represent the middle c or midi note number 60. Variety of scales comes packed with Panaeolus (major, minor, pentatonic, modal scales etc.). The abbreviation m4 means meter is 4/4 and the isolated '4' means the notes that come after it will be 4th part notes, so the number '8' would mean twice as fast note values, '16' twice as faster than '8' etc. Semitone transpositions of sharps and flats can be added with hashtag (#) for sharp and a b (b) for flat, placed in front of the roman numeric letters. Alongside the scale mode there is drum mode and chord mode. Drum mode is based on the letter 'x' for rhythmical composition, good for when pitch does not matter (ex. audio samples and/or in techno club music).

```
"drum m4 x r x x:2"
```

This pattern introduces rests 'r' and beat division ':2', so it will produce a pattern of something like  .
Chord mode offers quite accurate description of chords, but still the problem of voice spreading between octaves is still being worked on in Panaeolus. But as of today all basic chord structures can be placed on every position in all its inversions. With great inspiration from harmony analysis of Schönberg and my harmony classes in school, I designed it so that all capitalized roman numbers represent major chords and lower case represent minor chords. Suspended tones, sevenths and ninths can be added as well. The following pattern could clarify this a bit.

```
"c root=d3 m4 I6 iib7:2 IV46 #iv°56"
```



How these chord symbols are exactly translated to these chords is outside the scope of this paper. But the aim here is not to reinvent the wheel but to offer a notation that musicians understand and apply their knowledge from music schools into computer music.

6 Effects and .EvalCode

Adding effects to instruments in live performance can be solved in various ways. Since Csound does not still offer good way for variable parameters, then re-evaluating the whole instrument in performance can solve this. In Panaeolus, for every evaluation of pattern the instrument gets re-evaluated in Csound as well, irrelevant of if an effect was added or not. Adding effects to an instrument can be problematic if the live-coder wishes to have two pattern sending events to one instrument but adding effects to only one. For this reason a track number is concatenated into the instrument name in Csound. So for example, Csound instrument called `synth` then for track1 the instrument would get evaluated as `synth1` and track2 `synth2` etc. With this individuality between instruments and tracks, all effects or adjustments to instrument design can be made separate. Also worth pointing out the amount of possible instrument definitions in Csound is much higher than any ambitious live-coder could evaluate in one performance. A pseudo code for our dummy-instrument would look like this.

```
(defn orc-dummy [& {:keys [t-num fx-1 fx-2]
                    ;default arguments are empty strings
                    :or {t-num 1 fx-1 "" fx-2 ""}}]
  (let [str-> (apply str "
instr synth" t-num "\n
iamp = ampdb(p4)
ifreq = cpsmidinn(p5)
asig vco2 iamp, ifreq
aenv expon 1, p3, 0.1
afilt moogvcf2 asig, 310, 0.9
aM = afilt*aenv
" fx-1 "\n         ;mono audio signals to aM
aL = aM
aR = aM
" fx-2 "\n         ;stereo audio signals to aL, aR
outs aL, aR
endin")]
     (.EvalCode c str->)))
```

Note here that the API function .EvalCode is being used instead of .CompileOrc, this is the function that is found in CsoundQt when evaluate section command is used on instrument definition, a great function for when multiple re-evaluations on instrument is needed.

The function orc-dummy takes in 3 arguments, t-num for the track number, fx-1 for effects that operate on mono audio signals and fx-2 that operate on stereo audio signals. The design in Panaeolus (that does not have to be the best one) is that effects are stored in UDO and a callback to the UDO will be inserted to these fx-1/2 parameters and evaluated (on every track evaluation).

```
($1 (dummy-synth   :pat "s i iii v i'"
                  :dur [0.5 0.5 0.8]
                  :amp -10
                  :fx '((lo-fi :bit 8 :distortion 0.9)
                        ;A fictional Panaeolus style pattern with effects
                        (random-pan :rate 2))))
```

This fictional pattern would on evaluation immediately (re)evaluate the instrument in Csound, the string that is sent to the Csound instance with .EvalCode could look something like this (given that UDOs named lo-fi and random-pan have been compiled to the orchestra before).

```
instr synth3            ;if this were on track number 3
iamp = ampdb(p4)
ifreq = cpsmidinn(p5)
asig vco2 iamp, ifreq
aenv expon 1, p3, 0.1
afilt moogvcf2 asig, 310, 0.9
aM = afilt*aenv
aM lo-fi aM, 8, 0.9          ;callback to lo-fi has been added
aL = aM
aR = aM
aL, aR random-pan aL, aR, 2  ;callback to random-pan has been added
outs aL, aR
endin
```

Without going too deep into how this is done, it's enough to point out that each effect defined in UDO is also a Clojure function in Panaeolus where the effect function returns a string with given parameters and information of whether it's a mono or stereo effect. As the effects are not controlled with Csound parameters via score events, then as of now it's not possible to create iterating patterns on the effect parameters. But more research is needed on how to solve that.

Conclusion

Live-coding in Csound with the CsoundAPI offers the possibility of descriptive and powerful pattern generation by using general purpose programming language. Despite being focused on Clojure, the API is available to various other programming languages, opening the doors of live-coding program design impossible with the Csound language alone.

References

[1] Arthur Ulfeldt, February 2012, [online] http://stackoverflow.com/questions/9132346/clojure-differences-between-ref-var-agent-atom-with-examples, accessed 11 February 2016.