Exploring Melody Space in a Live Context Using Declarative Functional Programming

FARM Workshop at ICFP 2014, Gothenburg
Thomas G. Kristensen, uSwitch Ltd, London
Composer is a simple, responsive and extensible system utilising logic programming to allow novices to explore and learn music rules
Background
Background

offline

online
Background

programmers     musicians

offline

online
Background

programmers

musicians

offline

online
Background
Background

programmers

musicians

offline

Shasheela

FHarm

online

Principled approach
Background

- offline
  - programmers: Shasheela
  - musicians: FHarm
- online
  - programmers: Principled approach
  - musicians: Graphical score grammars
Background

Anders: Composing music by composing rules (Ph.D. thesis)
Koops, Magalhãe and de Haas: A functional approach to automatic melody harmonisation
Aaron, Blackwell, Hoadley and Regan: A principled approach to developing new languages for live coding
Stead, Blackwell and Aarong: Graphic score grammars for end-users
Melody rules

- Tonic note
- Mode
- Cadence
Melody rules

- Tonic note
- Mode
- Cadence
3.3 System architecture

The system itself is implemented in the functional programming language Clojure. Clojure was chosen based on its libraries available for doing logic programming (core.logic) and music synthesis (Overtone), and its strong concurrency support. Clojure is a hosted language, running on the Java Virtual Machine (JVM) platform. The COMPOSER program is implemented as a set of processes, which communicate through channels in a Communicating Sequential Processes (CSP) [10, 15] style, using the core.async library [2]. The general architecture is illustrated in Figure 3.

OSC messages are sent as UDP messages from the TouchOSC device to an OSC server running inside the JVM. The OSC listener is informed of OSC messages coming in. The listener takes the raw OSC messages, such as \{:path "/1/toggle8", :host "127.0.0.1", :args (1)} and translates them to domain-specific events, such as \{:tonic-note :C#3}. The domain events are put on a channel for the instrument state loop to consume.

The instrument state loop keeps track of the current state of the virtual instrument in the system. It consumes instrument updates and updates the instrument states, emitting new states on the instrument state channel. This channel in turn is broadcast onto two other channels which feed back into the OSC listener and the composer loop. The OSC listener feeds the instrument state back to the TouchOSC interface over UDP as OSC messages, to ensure that the performers view of the instrument state corresponds to the actual state.

The composer loop consumes instrument states, such as \{:tonic-note :C#3, :scale :major-scale, :cadence :perfect} and constructs a logic program that is evaluated using core.logic, to construct a stream of candidate melodies. A random melody is selected and emitted onto the channel feeding into the Overtone loop. The Overtone loop keeps track of which melody to play in the next iteration, and informs Overtone as to what to play and in what tempo. Overtone in turn co-ordinates the melody and synths with SuperCollider.

Much of the inter-process communication described above could have been achieved using direct process communication rather than channels. Channels were chosen for two reasons: (1) to decouple the processes, allowing for cleaner interfaces between components of the system; and (2), to control the stream instructions coming in from the user of the system. As demonstrated in the Experiments section, generating a large selection of melodies can be very time consuming, and the goal is for the experience from the user’s perspective to be as responsive as possible. If it takes more than a few seconds for a new composition to be generated and played, the instrument will not be suitable as an interactive performance tool.
is informed of OSC messages coming in. The listener takes the raw device to an OSC server running inside the JVM. The OSC listener processes, which communicate through channels in a Communicating Semaphore (CSP) style, using the core.async library [2]. The general architecture is illustrated in Figure 3.

The system itself is implemented in the functional programming language Clojure. Clojure was chosen based on its libraries available in the domain of music-synthesis, such as the library overtone [4].

Sequential Processes (CSP) [10, 15] style, using the core.async library, is used to handle the incoming messages directly. In this way, each incoming message is consumed in isolation, allowing for parallel execution.

The Overtone loop keeps track of which melody to play in the next iteration, and informs Overtone as to what to play and in what tempo. Overtone in turn co-ordinates the melody and synthesizers, constructing a stream of candidate melodies. A random melody is selected and emitted onto the channel feeding into the Overtone server.

The composer loop consumes instrument states, such as the current state of the virtual instrument in the system. It consumes instrument updates and translates them to domain-specific events, such as a melody being selected or a scale being chosen.

The composer loop is decoupled from the rest of the system, allowing for cleaner interfaces between the processes. As demonstrated in the Experiments section, generating a large selection of melodies can be very time consuming, and the goal is for the experience from using the system to be suitable as an interactive performance tool.

The row at the bottom left is used to choose a mode or scale from left to right: major scale, harmonic minor scale, natural minor scale, locrian mode, and mixolydian mode. A single mode is selected and emitted onto the channel feeding into the Overtone server.

A button at the bottom right of the interface consists of two pages. The second page of the interface consists of two pages. The second page of the interface has a section for changing the tempo of the melodies being played. The current tempo is displayed on the left side of the page.

The sliders to the right controls the individual relative pauses between notes being played. The pauses are normalised to always keep the slider to the left controls the beats-per-minute (BPM), and the eight sliders to the right controls the beats-per-minute of the individual notes.

The pauses are normalised to always keep the BPM constant. The interface is designed to allow for easy adjustment of the tempo, with the BPM being set to the desired value by moving the slider to the left. The BPM is displayed on the left side of the page.

The row at the bottom right of the interface is used to choose a cadence. The cadences in order are: perfect, plagal and un-named.

The interface consists of two pages. The second page of the interface has a section for changing the tempo of the melodies being played. The current tempo is displayed on the left side of the page.

The row at the bottom left is used to choose a mode or scale from left to right: major scale, harmonic minor scale, natural minor scale, locrian mode, and mixolydian mode. A single mode is selected and emitted onto the channel feeding into the Overtone server.

A button at the bottom right of the interface consists of two pages. The second page of the interface has a section for changing the tempo of the melodies being played. The current tempo is displayed on the left side of the page.

The sliders to the right controls the individual relative pauses between notes being played. The pauses are normalised to always keep the BPM constant. The interface is designed to allow for easy adjustment of the tempo, with the BPM being set to the desired value by moving the slider to the left. The BPM is displayed on the left side of the page.

The row at the bottom right of the interface is used to choose a cadence. The cadences in order are: perfect, plagal and un-named.
Logic programming
Logic programming

(run* [notes]
    (scaleo :C3 major-scale notes)
    (counto notes 8))

Logic programming

```
(run* [notes]
  (scaleo :C3 major-scale notes)
  (counto notes 8))

(run 3 [m1 m2 m3 m4 m5 m6 m7 m8]
  (fresh [n1 n2 n3 n4 n5 n6 n7 n8]
    (scaleo :C3 major-scale
      [n1 n2 n3 n4 n5 n6 n7 n8])
    (permuteo [m1 m2 m3 m4 m5 m6 m7 m8]
      [n1 n2 n3 n4 n5 n6 n7 n8])
    (== m1 :C3)
    (== m8 :C4)))
```
Logic programming

(run* [notes]
  (scaleo :C3 major-scale notes)
  (counto notes 8))

(run 3 [m1 m2 m3 m4 m5 m6 m7 m8]
  (fresh [n1 n2 n3 n4 n5 n6 n7 n8]
    (scaleo :C3 major-scale
      [n1 n2 n3 n4 n5 n6 n7 n8])
    (permuteo [m1 m2 m3 m4 m5 m6 m7 m8]
      [n1 n2 n3 n4 n5 n6 n7 n8])
    (== m1 :C3)
    (== m8 :C4)))

(run* [tonic-note pattern]
  (scaleo tonic-note pattern
  ;; => ([:C3 (1 0 1 0 1 1 0 1 0 1 0 1 1 . _0)])
Logic programming

(ns composer.composer
  (:refer-clojure :exclude [==])
  (:require [clojure.core.async :refer [go >! <!]]
            [clojure.core.logic :refer :all]
            [clojure.core.logic.plldb :refer :all]))

(defn scale-from-tones [tone-types]
  (take 25
    (->> tone-types
         (map {:semitone [1]
              :tone [0 1]
              :minor-third [0 0 1]})
         flatten
         butlast
         (cons 1)
         cycle)))


(def scale-modes
  [:major-scale :minor-scale :harmonic-minor-scale :natural-minor-scale :locrian-mode :mixolydian-mode])

(def semitone-facts
  (reduce
   (fn [db [note-1 note-2]]
     (db-fact db semitone note-1 note-2))
   empty-db
   (partition 2 1 keys-from-c)))

(def scaleo [base-note scale notes]
  ([note [1 . scale-rest] [note . ()]]
   ([note [1 . scale-rest] [note . notes-rest]]
     (fresh [next-note]
       (semitone note next-note)
       (scaleo next-note scale-rest notes-rest))))

(def booleans [:true :false])

(defn key-restriction [instrument-state s1]
  (if-let [key (:key instrument-state)]
    (all (== key s1))
    succeed))

(defn scale-restriction [instrument-state scale-type]
  (if (:scale instrument-state)
    (all (membero [:scale instrument-state] scale-modes))
    succeed))

(defn cadence-restriction [instrument-state m7 s2 s4 s5]
  (case (:cadence instrument-state)
    :perfect   (all (== m7 s5))
    :plagal    (all (== m7 s4))
    :just-nice (all (== m7 s2))
    nil        succeed))

(defn- logic-program [instrument-state melody2]
  (fresh [melody
          m1 m2 m3 m4 m5 m6 m7 m8
          scale
          s1 s2 s3 s4 s5 s6 s7 s8
          base-note scale-type
          key-restriction instrument-state s1]
    (= melody [m1 m2 m3 m4 m5 m6 m7 m8]
              (= scale [s1 s2 s3 s4 s5 s6 s7 s8]
                      (= m1 s1)
                      (= m2 s2)
                      (= m3 s3)
                      (= m4 s4)
                      (= m5 s5)
                      (= m6 s6)
                      (= m7 s7)
                      (= m8 s8)
                      (= cadence-restriction instrument-state m7 s2 s4 s5)
                      (= melody2 [m1 m2 m3 m4 m5 m6 m7 m8]
                                  (= scale-restriction instrument-state scale-type)
                                  (= permutate scale melody))
    (= composition [instrument-state] melody2)
    (= melody (apply dissoc instrument-state scale))
    (= melody2 (apply dissoc instrument-state scale)))
  )

(defn- random-composition [instrument-state]
  (rand-nth
    (or (seq (compositions instrument-state 1024))
        [])))

(defn composer-loop
  "Listens for new instrument states on instrument-state-ch and emits a random melody to melody-ch. The loop terminates when instrument-state-ch closes.
   Changes to :speed or :gaps does not compose a new melody, but alters the timing of the existing."
  [instrument-state-ch melody-ch]
  (go
    (loop [prev-instrument-state nil
           prev-composition nil]
      (when-let [instrument-state (<! instrument-state-ch)]
        (let [gaps (for [i (range 8)] (get (:gaps instrument-state) i 0.5))
              speed (:speed instrument-state)
              new-melody (if (same-melody-params? prev-instrument-state
                                                    instrument-state)
                            (:melody prev-composition)
                            (random-composition instrument-state))]
          (= melody (apply dissoc instrument-state scale))
          (= melody2 (apply dissoc instrument-state scale-type))
          (= scale-restriction instrument-state scale-type)
          (= melody (apply dissoc instrument-state scale))
          (= permutate scale melody))))

(defn compositions [instrument-state & [n]]
  (with-db semitone-facts
    (if n
      (run n [melody2]
        (logic-program instrument-state melody2))
      (run* [melody2]
        (logic-program instrument-state melody2)))))
The system
The system
Experiments

• Goal: a reactive system

• Experiment 1: What is the size of the melody space and how long does it take to enumerate it?

• Experiment 2: What is a reasonable bound on the search space to achieve responsiveness?
## Experiment I

<table>
<thead>
<tr>
<th></th>
<th>No scale</th>
<th></th>
<th>Major scale</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Any tonic note</td>
<td>$C$</td>
<td>Any tonic note</td>
<td>$C$</td>
</tr>
<tr>
<td></td>
<td>$25^8$</td>
<td>$25^8$</td>
<td>$9,360$</td>
<td>$720$</td>
</tr>
<tr>
<td>Melody space</td>
<td>$25^8$</td>
<td>$25^8$</td>
<td>$1,560$</td>
<td>$120$</td>
</tr>
<tr>
<td>Execution time (ms)</td>
<td>$-$</td>
<td>$-$</td>
<td>$4,299$</td>
<td>$294$</td>
</tr>
<tr>
<td>Melodies/second</td>
<td>$-$</td>
<td>$-$</td>
<td>$3,852$</td>
<td>$278$</td>
</tr>
<tr>
<td></td>
<td>$2,177$</td>
<td>$404$</td>
<td>$2,448$</td>
<td>$431$</td>
</tr>
</tbody>
</table>
Experiment 2

- No scale
- Major scale, any tonic note, any cadence
- Major scale, C, any cadence
- Major scale, C, perfect cadence

Graph showing execution time (ms) versus search space bound.
Conclusion

- Composer demonstrates it is possible to build a responsive interactive system with extremely small and succinct core

- The declarative nature of the core implementation makes it possible to extend the terminology to other types of music
Future work

• Proper sampling of search space
• Labeled interface
• Non-Western music
• User testing