The Arpeggigon: A Functional Reactive Musical Automaton

Haskell eXchange 2017, 13 Oct., London

Henrik Nilsson

Joint work with Guerric Chupin and Jin Zhan

Functional Programming Laboratory, School of Computer Science

University of Nottingham, UK

The Arpeggigon (1)

- Software realisation of the reacTogon:

- Interactive cellular automaton:
  - Configuration
  - Performance parameters

The Arpeggigon (2)

- Implemented in Haskell using:
  - The Functional Reactive Programming (FRP) system Yampa
  - Reactive Values and Relations (RVR)
- Based on the *Harmonic Table*

Code: `https://gitlab.com/chupin/arpeggigon`
Video:
`https://www.youtube.com/watch?v=v0HIkFR1EN4`

Before you get too excited: ***Work in progress!***

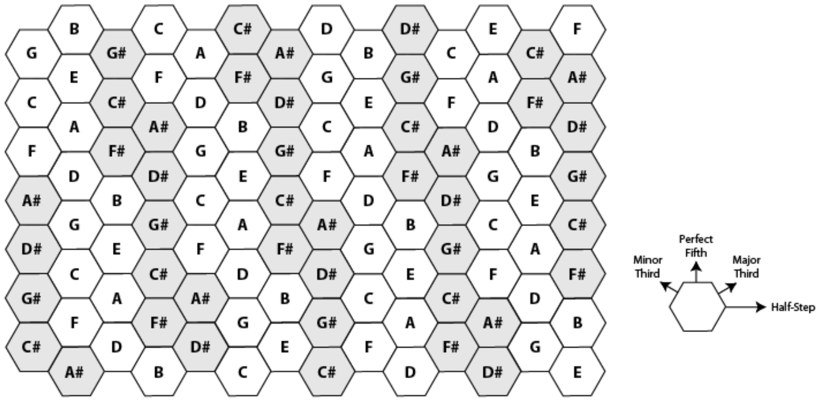
Motivation

Exploring FRP and RVR as an (essentially) declarative way for developing full-fledged musical applications:

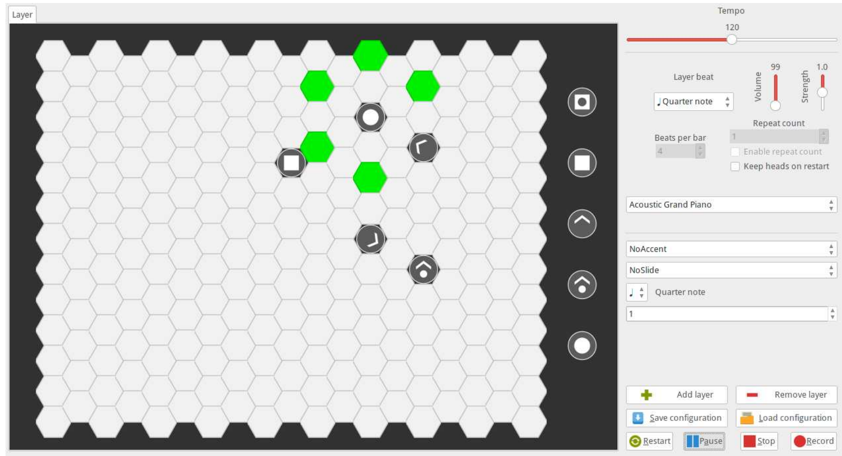
- FRP aligns with declarative and temporal (discrete and continuous) nature of music
- RVR allows declarative-style interfacing with external components

The ***structure*** of the application should be such that it in principle is usable in a MIDI-studio setting.

The Harmonic Table



Running a Sample Configuration

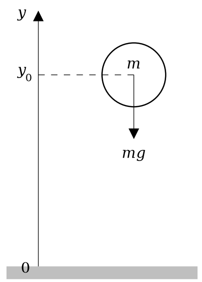


The Rest of this Talk

- Brief introduction to FRP and Yampa
- The Arpeggigon core
- Brief introduction to Reactive Values and Relations
- The Arpeggigon shell

Functional Reactive Programming (1)

- Key idea: Don't program one-time-step-at-a-time, but describe an evolving entity as whole.



$$y = y_0 + \int v \, dt$$

$$v = v_0 + \int -9.81$$

We are used to describing behaviours in totallity over time in mathematics. Why not program in the same way?

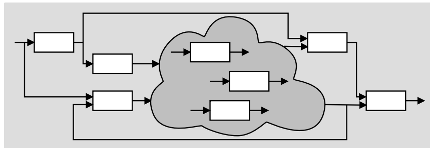
Functional Reactive Programming (2)

Combines conceptual simplicity of the synchronous data flow approach with the flexibility and abstraction power of higher-order functional programming:

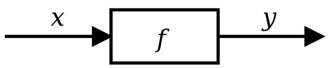
- Synchronous
- First class temporal abstractions
- Hybrid: mixed continuous and discrete time
- Dynamic system structure

Good conceptual fit for many applications, including games and, as we will see here, interactive musical applications.

Yampa

- FRP implementation embedded in Haskell
- Key notions:
  - ***Signals***: time-varying values
  - ***Signal Functions***: pure functions on signals
  - ***Switching***: temporal composition of signal functions
- Programming model: 

Signal Functions



Intuition:

$$\begin{array}{l} \mathit{Time} \approx \mathbb{R} \\ \mathit{Signal}\ a \approx \mathit{Time} \rightarrow a \\ x :: \mathit{Signal}\ \mathit{T1} \\ y :: \mathit{Signal}\ \mathit{T2} \\ \mathit{SF}\ a\ b \approx \mathit{Signal}\ a \rightarrow \mathit{Signal}\ b \\ f :: \mathit{SF}\ \mathit{T1}\ \mathit{T2} \end{array}$$

Additionally, ***causality*** required: output at time t must be determined by input on interval $[0, t]$.

Some Basic Signal Functions

$$\mathit{identity} :: \mathit{SF}\ a\ a$$

$$\mathit{constant} :: b \rightarrow \mathit{SF}\ a\ b$$

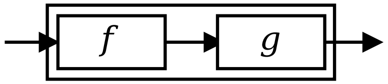
$$\mathit{integral} :: \mathit{VectorSpace}\ a\ s \Rightarrow \mathit{SF}\ a\ a$$

$$y(t) = \int_0^t x(\tau)\,\mathrm{d}\tau$$

Composition

In Yampa, systems are described by combining signal functions (forming new signal functions).

For example, serial composition:

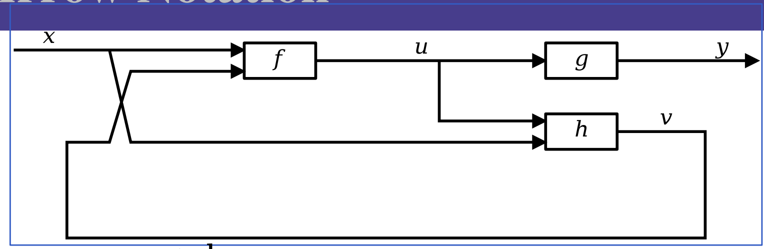


A ***combinator*** that captures this idea:

$$(\ggg) :: SF\ a\ b \rightarrow SF\ b\ c \rightarrow SF\ a\ c$$

Signal functions are the primary notion; signals a secondary one, only existing indirectly.

Arrow Notation



$$\begin{array}{l} \textbf{proc}\ x \rightarrow \textbf{do} \\ \quad \textbf{rec} \\ \qquad u \leftarrow f \prec (x, v) \\ \qquad y \leftarrow g \prec u \\ \qquad v \leftarrow h \prec (u, x) \\ \quad returnA \prec y \end{array}$$

Only syntactic sugar: everything translated into a combinator expression.

Events

Yampa models discrete-time signals by lifting the ***co-domain*** of signals using an option-type:

$$\textbf{data}\ Event\ a = NoEvent \mid Event\ a$$

Discrete-time signal = `Signal` (`Event` α).

Some functions and event sources:

$$tag :: Event\ a \rightarrow b \rightarrow Event\ b$$

$$after :: Time \rightarrow b \rightarrow SF\ a\ (Event\ b)$$

$$edge :: SF\ Bool\ (Event\ ())$$

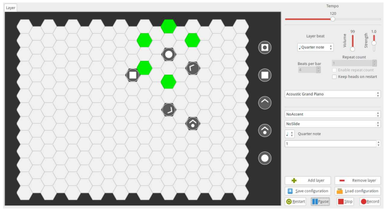
Switching

Idea:

- Allows one signal function to be replaced by another.
- Switching takes place on the first occurrence of the switching event source.

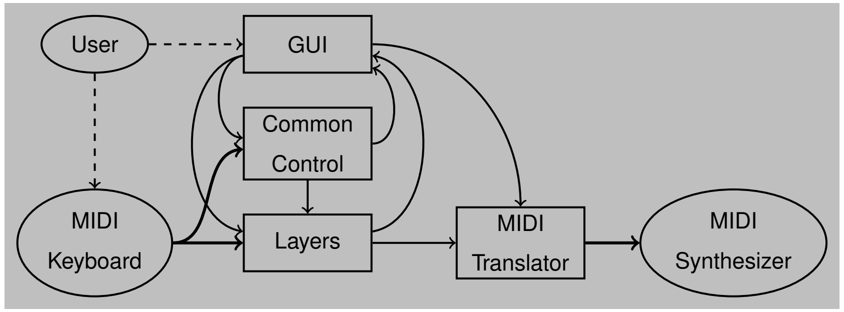
$$\begin{array}{l} switch :: \\ \quad SF\ a\ (b, Event\ c) \\ \quad \rightarrow (c \rightarrow SF\ a\ b) \\ \quad \rightarrow SF\ a\ b \end{array}$$

Aspects of the Arpeggigon



- ***Interactive***
- Layers can be added/removed: ***dynamic structure***
- Notes generated at ***discrete*** points in time
- Configuration and performance parameters can be changed at ***any*** time

Arpeggigon Architecture



Cellular Automaton

State transition function for the cellular automaton:

$$
\begin{aligned}
&\mathit{advanceHeads} :: \mathit{Board} \to \mathit{BeatNo} \to \mathit{RelPitch} \to \mathit{Strength} \\
&\qquad\qquad \to [\mathit{PlayHead}] \to ([\mathit{PlayHead}], [\mathit{Note}])
\end{aligned}
$$

Lifted into a signal function primarily using $\mathit{accumBy}$:

$$\mathit{accumBy} :: (b \to a \to b) \to b \to \mathit{SF}\ (\mathit{Event}\ a)\ (\mathit{Event}\ b)$$

$$
\begin{aligned}
&\mathit{automaton} :: [\mathit{PlayHead}] \\
&\qquad \to \mathit{SF}\ (\mathit{Board}, \mathit{DynamicLayerCtrl}, \mathit{Event}\ \mathit{BeatNo}) \\
&\qquad\qquad (\mathit{Event}\ [\mathit{Note}], [\mathit{PlayHead}])
\end{aligned}
$$

Automated Smooth Tempo Change

Smooth transition between two preset tempos:

$$
\begin{aligned}
&\mathit{smoothTempo} :: \mathit{Tempo} \to \mathit{SF}\ (\mathit{Bool}, \mathit{Tempo}, \mathit{Tempo}, \mathit{Rate})\ \mathit{Tempo} \\
&\mathit{smoothTempo}\ \mathit{tpo0} = \mathbf{proc}\ (\mathit{sel1}, \mathit{tpo1}, \mathit{tpo2}, \mathit{rate}) \to \mathbf{do} \\
&\quad \mathbf{rec} \\
&\qquad \mathbf{let}\ \mathit{desTpo} = \mathbf{if}\ \mathit{sel1}\ \mathbf{then}\ \mathit{tpo1}\ \mathbf{else}\ \mathit{tpo2} \\
&\qquad\quad\ \ \mathit{diff} \quad\ \ = \mathit{desTpo} - \mathit{curTpo} \\
&\qquad\quad\ \ \mathit{rate}' \quad = \mathbf{if}\ \quad\ \ \mathit{diff} > 0.1\ \ \ \mathbf{then}\ \mathit{rate} \\
&\qquad\qquad\qquad\quad\ \ \mathbf{else\ if}\ \mathit{diff} < -0.1\ \mathbf{then}\ -\mathit{rate} \\
&\qquad\qquad\qquad\quad\ \ \mathbf{else}\qquad\qquad\qquad\quad 0 \\
&\qquad \mathit{curTpo} \leftarrow \mathit{arr}\ (+\mathit{tpo0}) \lll \mathit{integral} \prec \mathit{rate}' \\
&\quad \mathit{returnA} \prec \mathit{curTpo}
\end{aligned}
$$

Reactive Values and Relations (1)

- The Arpeggigon interacts with the outside world using two imperative toolkits:
  - GUI: GTK+
  - MIDI I/O: Jack
- Very imperative APIs: Hard or impossible to provide FRP wrappers.
- Instead, we use Ivan Perez's ***Reactive Values and Relations*** (RVR) to wrap the FRP core in a "shell" that acts as a bridge between the outside world and the pure FRP core.

Reactive Values and Relations (2)

- A Reactive Value (RV) is a typed mutable value with access rights and subscribable change notification.
- RVs provide a uniform interface to GUI widgets, files, network devices, . . .

  For example, the text field of a text input widget becomes an RV.
- Reactive Relations (RR) allow RVs to automatically be kept in synch by specifying the relations that should hold between them.

Reactive Values and Relations (3)

- While the RVR programming takes place in the IO monad, the code reads fairly declaratively as it specifies an interconnected network of RVs.
- Of course, RVR bindings need to be written for libraries that we wish to use unless available. Inevitably imperative code.
- RVR bindings for GTK+ are available; Jack bindings were written from scratch.

System Tempo Slider

```
globalSettings :: IO (VBox, ReactiveFieldReadWrite IO Int)
globalSettings = do
  globalSettingsBox ← vBoxNew False 10
  tempoAdj          ← adjustmentNew 120 40 200 1 1 1
  tempoLabel        ← labelNew (Just "Tempo")
  boxPackStart globalSettingsBox tempoLabel PackNatural 0
  tempoScale        ← hScaleNew tempoAdj
  boxPackStart globalSettingsBox tempoScale PackNatural 0
  scaleSetDigits tempoScale 0
  let tempoRV =
    bijection (floor, fromIntegral)
    `liftRW` scaleValueReactive tempoScale
  return (globalSettingsBox, tempoRV)
```

Pause

- Pausing is achieved by setting the tempo to 0 when the pause button is engaged.
- Easy to implement by combining two RVs:

$$
\begin{aligned}
&\mathit{tempoRV}' = \\
&\quad \mathit{liftR2}\ (\lambda \mathit{tempo}\ \mathit{paused} \rightarrow \textbf{if}\ \mathit{paused}\ \textbf{then}\ 0\ \textbf{else}\ \mathit{tempo}) \\
&\qquad\qquad \mathit{tempoRV} \\
&\qquad\qquad \mathit{pauseButtonRV}
\end{aligned}
$$

- This is an equation defining $\mathit{tempoRV}'$ once and for all.

Connecting the Core to the Shell

The following function makes a signal function available as RVs:

$$
\begin{aligned}
&\mathit{yampaReactiveDual} :: \\
&\quad a \\
&\quad \rightarrow \mathit{SF}\ a\ b \\
&\quad \rightarrow \mathit{IO}\ (\mathit{ReactiveFieldWrite}\ \mathit{IO}\ a, \mathit{ReactiveFieldRead}\ \mathit{IO}\ b)
\end{aligned}
$$

This creates two reactive values: one for the input and one for the output of the signal function. After writing a value to the input, the corresponding output at that point in time can be read.

Summary

- Yampa (FRP) good fit for writing interactive musical applications in a declarative way.
- Reactive Values and Relations proved very helpful for bridging the gap between the outside world and the FRP core in a fairly declarative way.
- Performance in terms of overall execution time and space perfectly fine.
- ***Timing*** is not yet as tight as it should be due to naive MIDI generation.

Reading (1)

- Henrik Nilsson and Guerric Chupin. Funky Grooves: Declarative Programming of Full-Fledged Musical Applications. In *9th International Symposium on Practical Aspects of Declarative Languages (PADL 2017)*, pp. 163–172, January 2017.
- Ivan Perez and Henrik Nilsson. Bridging the GUI Gap with Reactive Values and Relations. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell (Haskell'15)*, pp. 47–58, September 2015.

Reading (2)

- Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 Haskell Workshop*, pp. 51–64, October 2002.
- Antony Courtney and Henrik Nilsson and John Peterson. The Yampa Arcade. In *Proceedings of the 2003 Haskell Workshop*, pp. 7–18, August 2003.