The Arpeggigon: A Functional Reactive Musical Automaton

Haskell in Leipzig 2017, 26–27 Oct., Leipzig

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:

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)

The Arpeggigon (2)

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

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`

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

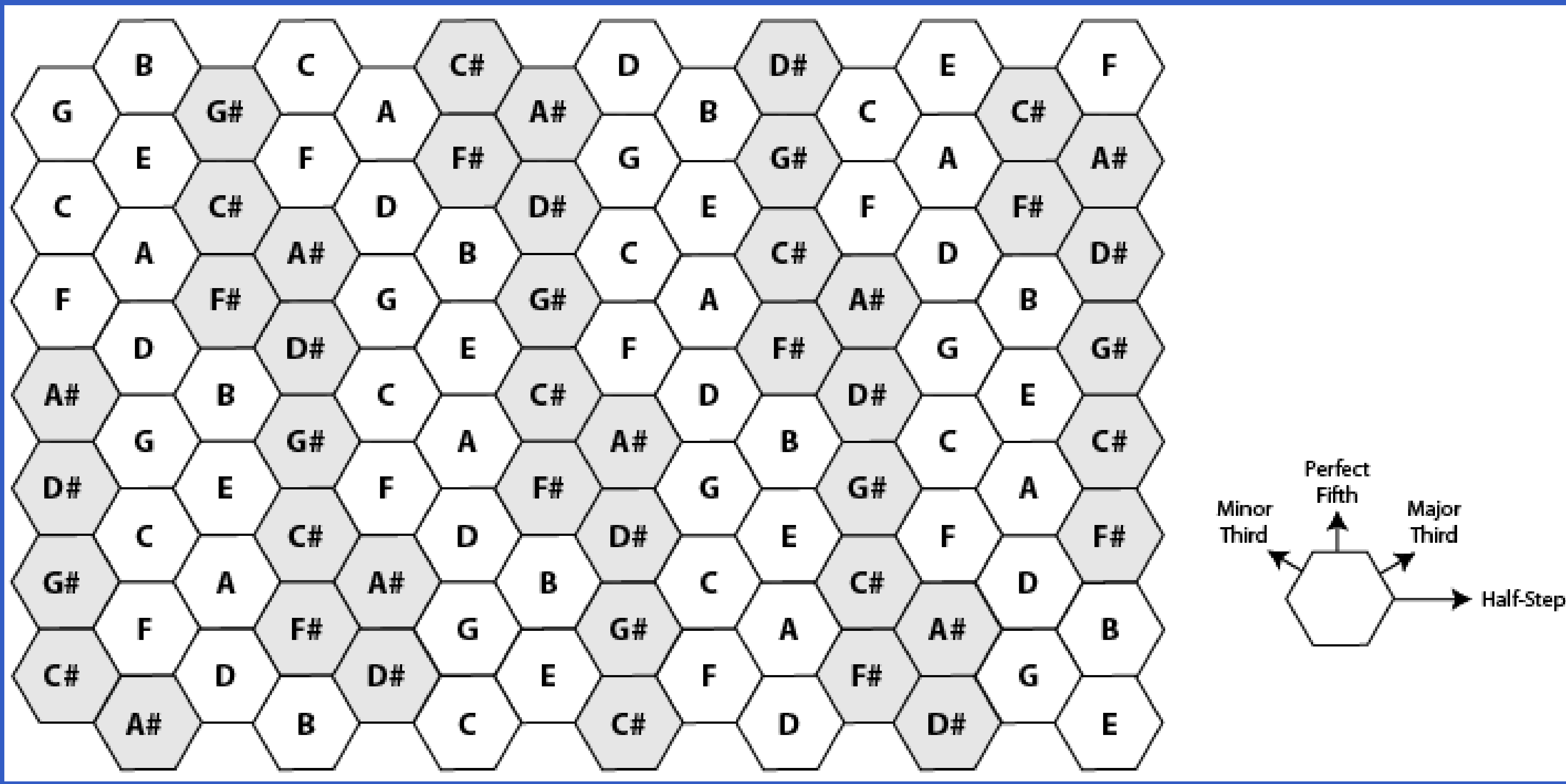
Motivation

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

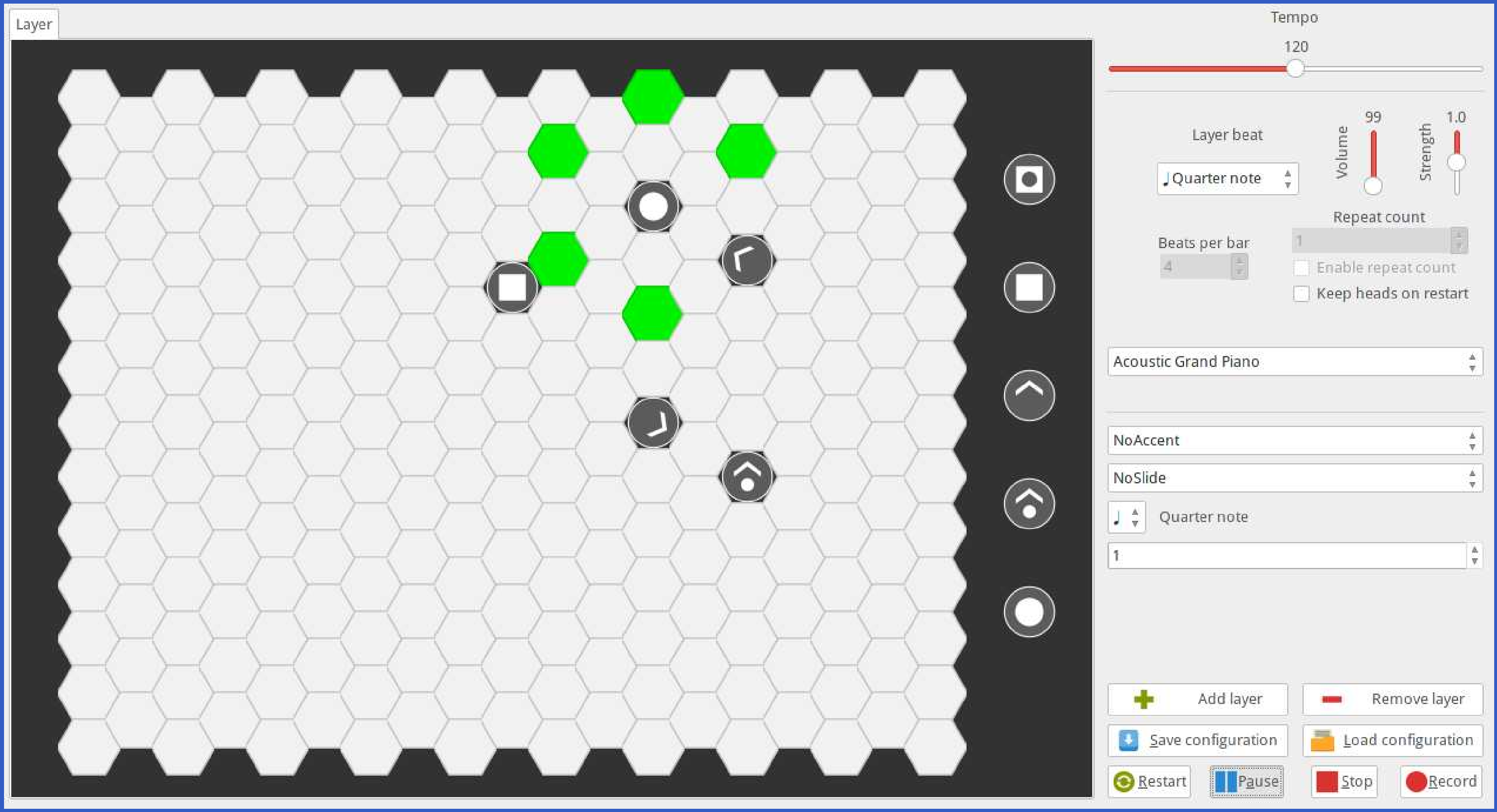
- FRP aligns with declarative and temporal (discrete and continuous) nature of music
- RVR allows declarative-style interfacing with external components

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

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

Functional Reactive Programming

- Key idea: Don't program one-time-step-at-a-time, but describe an evolving entity as a ***whole***.
- Combines conceptual simplicity of ***synchronous data flow*** with the flexibility of ***higher-order functional programming***:

Functional Reactive Programming

- Key idea: Don't program one-time-step-at-a-time, but describe an evolving entity as a ***whole***.
- Combines conceptual simplicity of ***synchronous data flow*** with the flexibility of ***higher-order functional programming***:
  - First class temporal abstractions

Functional Reactive Programming

- Key idea: Don't program one-time-step-at-a-time, but describe an evolving entity as a ***whole***.
- Combines conceptual simplicity of ***synchronous data flow*** with the flexibility of ***higher-order functional programming***:
  - First class temporal abstractions
  - Dynamic system structure

Functional Reactive Programming

- Key idea: Don't program one-time-step-at-a-time, but describe an evolving entity as a ***whole***.
- Combines conceptual simplicity of ***synchronous data flow*** with the flexibility of ***higher-order functional programming***:
  - First class temporal abstractions
  - Dynamic system structure
- Traditionally ***hybrid***: mixed continuous and discrete time

Functional Reactive Programming

- Key idea: Don't program one-time-step-at-a-time, but describe an evolving entity as a ***whole***.
- Combines conceptual simplicity of ***synchronous data flow*** with the flexibility of ***higher-order functional programming***:
  - First class temporal abstractions
  - Dynamic system structure
- Traditionally ***hybrid***: mixed continuous and discrete time

Good conceptual fit for games, musical applications . . .

Yampa

Yampa

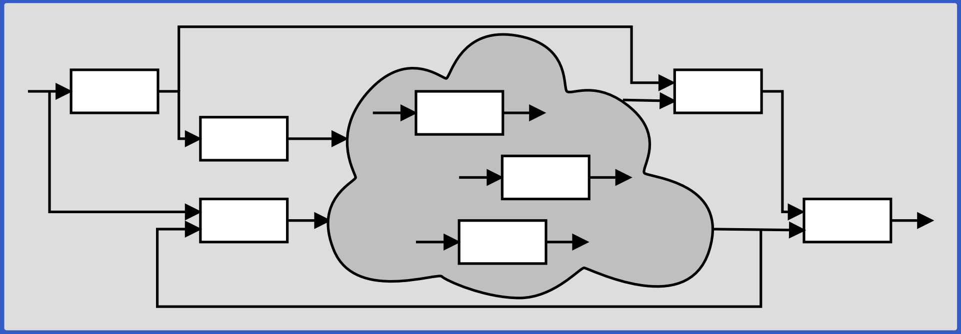
- FRP implementation embedded in Haskell

Yampa

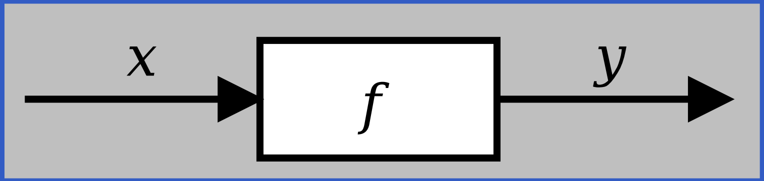
- FRP implementation embedded in Haskell
- Key notions:
  - ***Signals***: time-varying values
  - ***Signal Functions***: pure functions on signals
  - ***Switching***: temporal composition of signal functions

Yampa

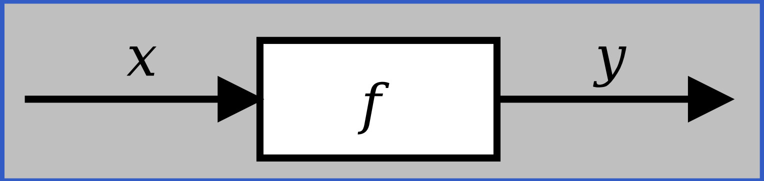
- FRP implementation embedded in Haskell
- Key notions:
  - ***Signals***: time-varying values
  - ***Signal Functions***: pure functions on signals
  - ***Switching***: temporal composition of signal functions
- Programming model:



Signal Functions (1)

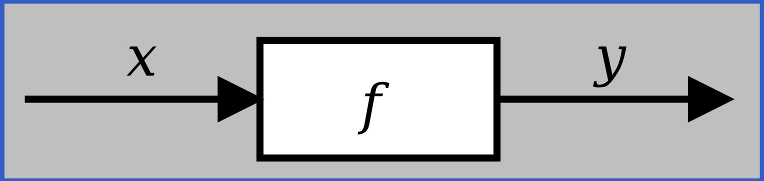


Signal Functions (1)



Intuition:

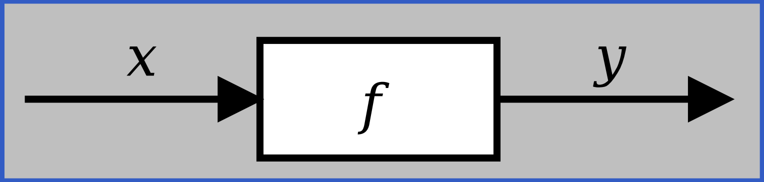
Signal Functions (1)



Intuition:

$$Time \approx \mathbb{R}$$

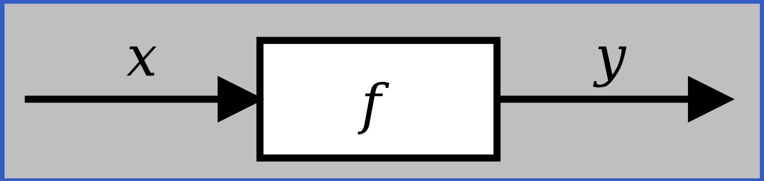
Signal Functions (1)



Intuition:

$$
\begin{aligned}
&Time \approx \mathbb{R} \\
&Signal\ a \approx Time \rightarrow a \\
&x :: Signal\ T1 \\
&y :: Signal\ T2
\end{aligned}
$$

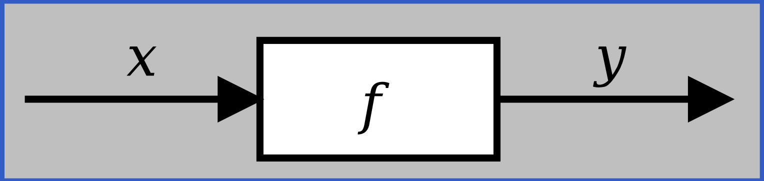
Signal Functions (1)



Intuition:

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

Signal Functions (1)

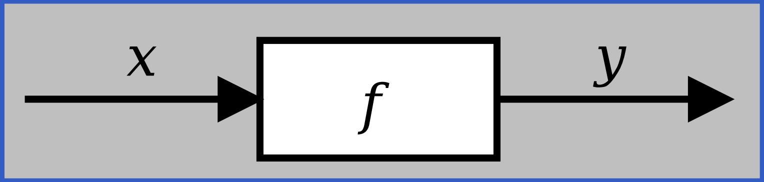


Intuition:

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

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

Signal Functions (2)



Example:

$$integral :: VectorSpace\ a\ s \Rightarrow SF\ a\ a$$

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

Clearly causal: output at time t determined by input on interval $[0, t]$.

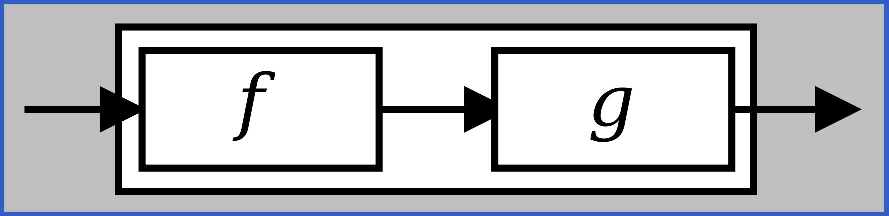
Composition

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

Composition

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

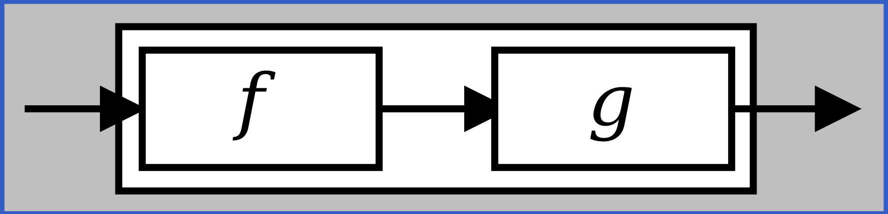
For example, serial composition:



Composition

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

For example, serial composition:



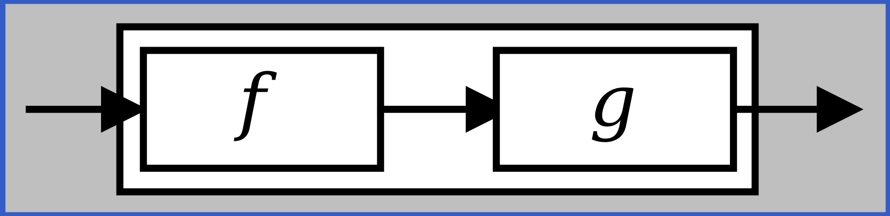
A ***combinator*** that captures this idea:

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

Composition

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

For example, serial composition:

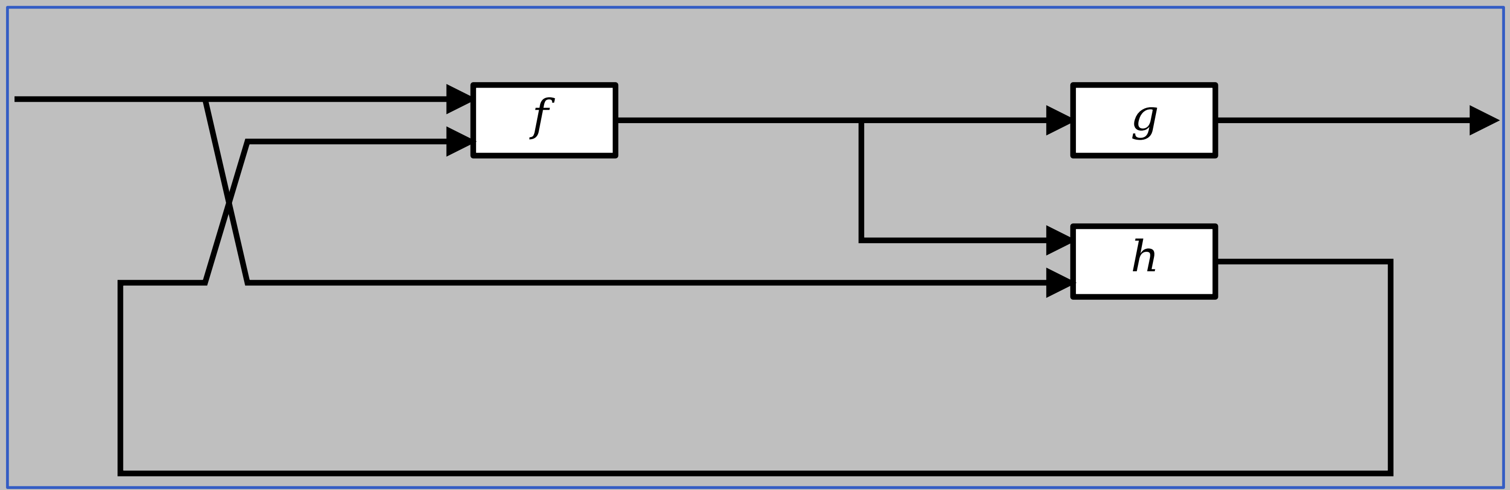


A ***combinator*** that captures this idea:

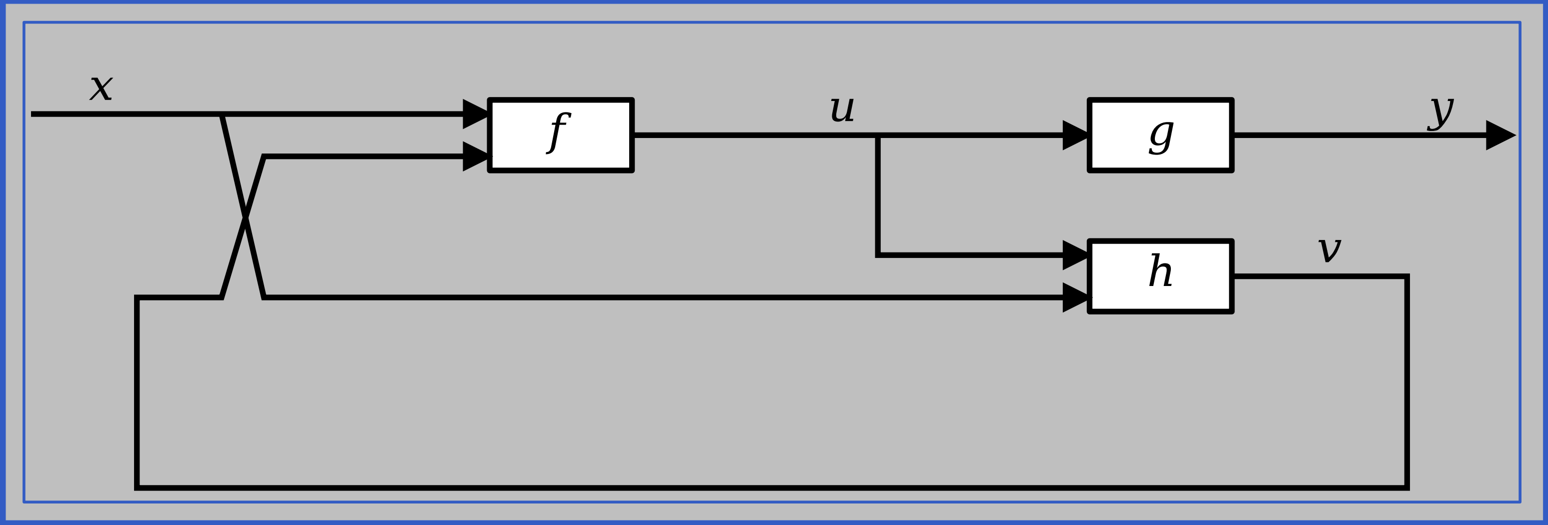
$$(\ggg) :: SF\ a\ b \rightarrow SF\ b\ c \rightarrow SF\ a\ c$$

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

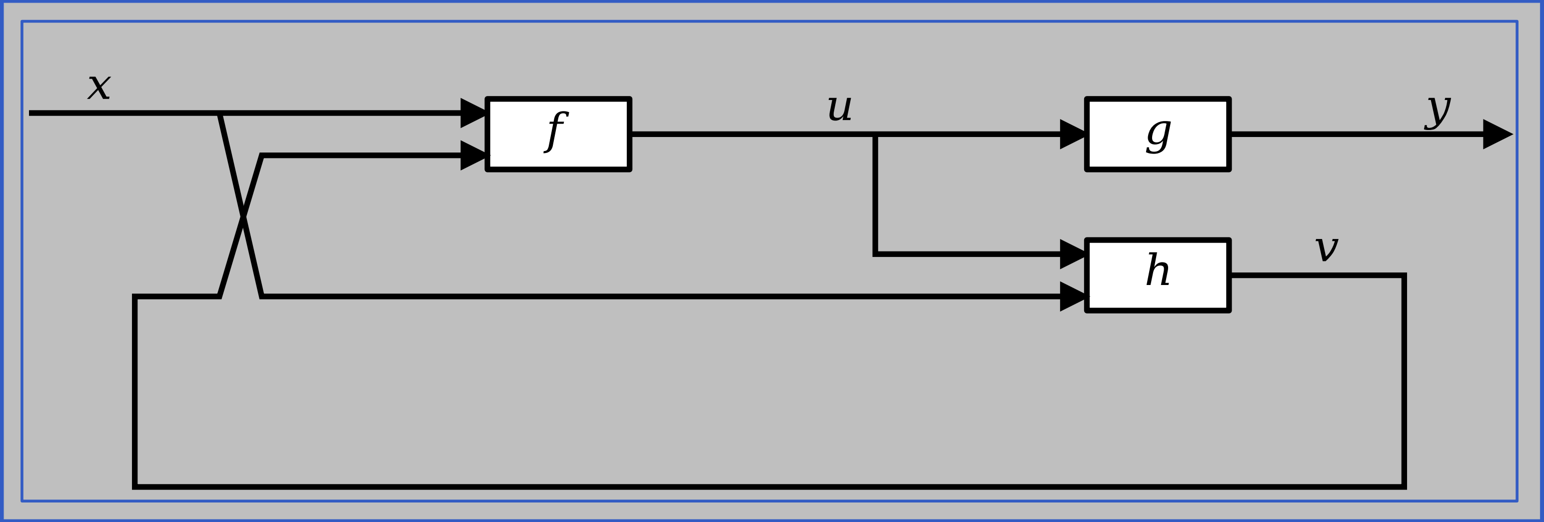
Arrow Notation



Arrow Notation



Arrow Notation



$\mathbf{proc}\ x \rightarrow \mathbf{do}$

$\quad \mathbf{rec}$

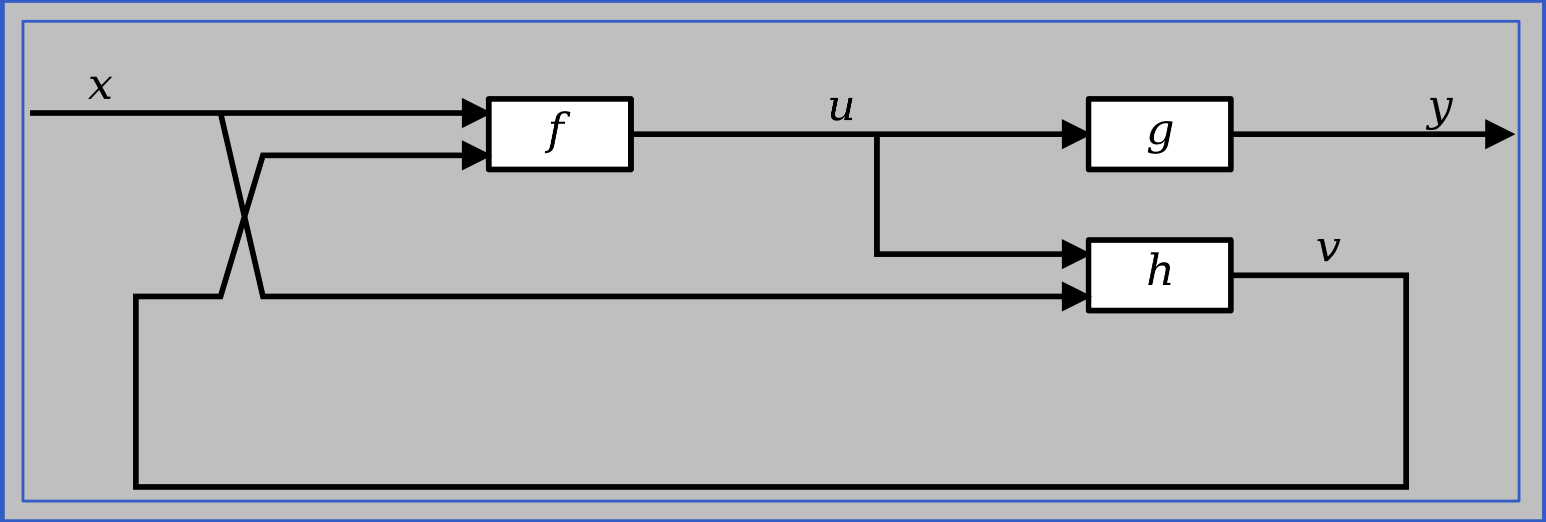
$\qquad u \leftarrow f \prec (x, v)$

$\qquad y \leftarrow g \prec u$

$\qquad v \leftarrow h \prec (u, x)$

$\quad returnA \prec y$

Arrow Notation



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

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:

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

Discrete-time signal $= \texttt{Signal}\ (\texttt{Event}\ \alpha)$.

Events

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

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

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

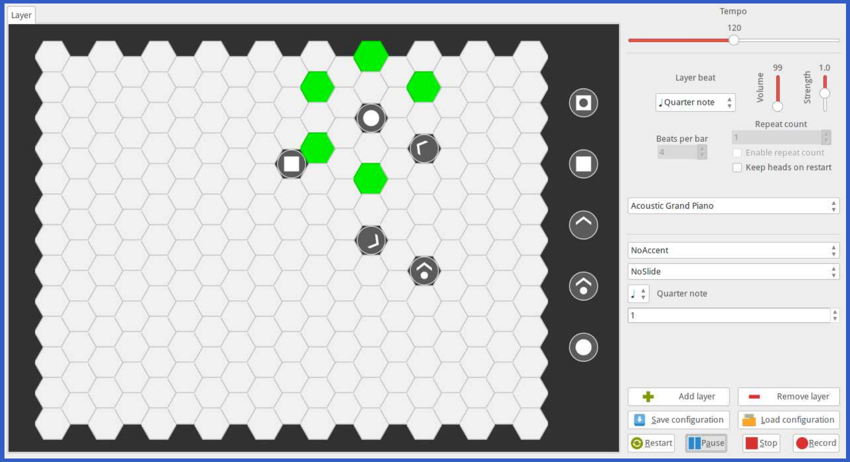
Some functions and event sources:

$$tag :: Event\ a \to b \to Event\ b$$

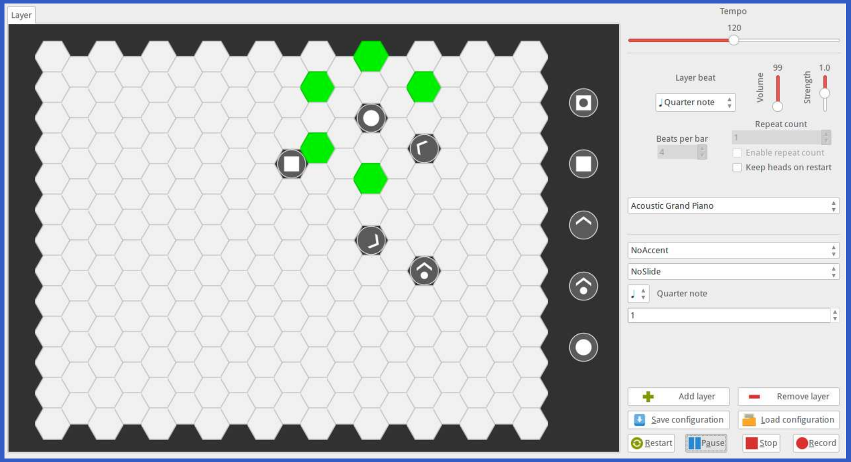
$$after :: Time \to b \to SF\ a\ (Event\ b)$$

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

Aspects of the Arpeggigon

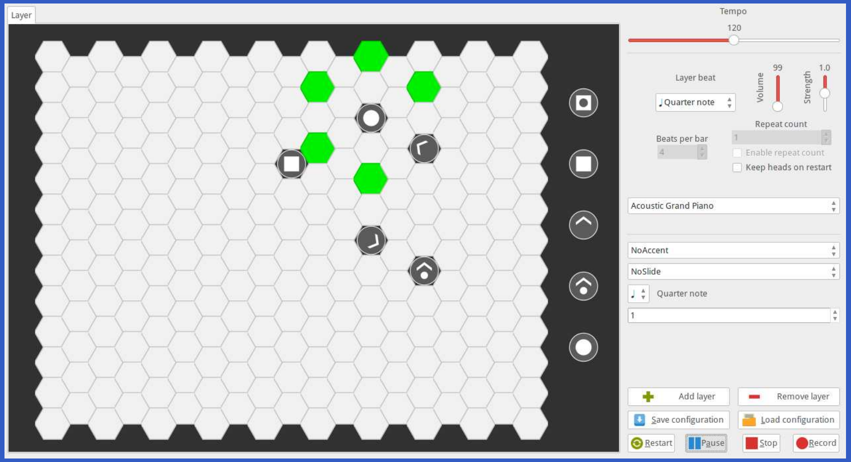


Aspects of the Arpeggigon



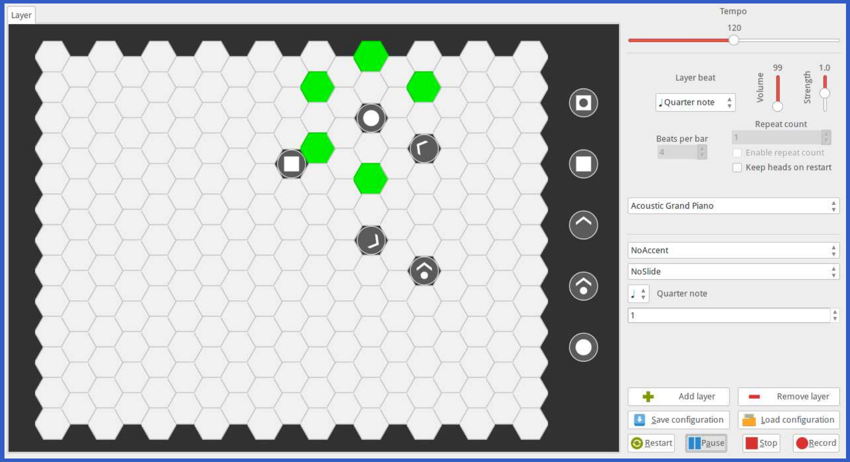
- ***Interactive***

Aspects of the Arpeggigon



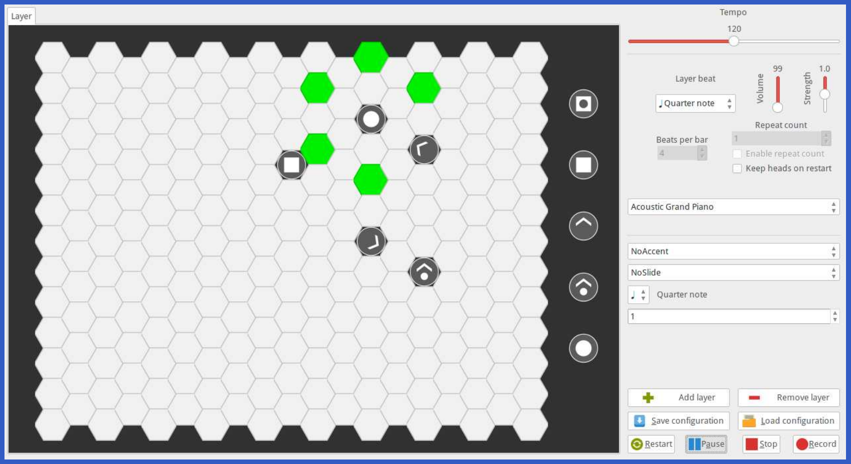
- ***Interactive***
- Layers can be added/removed: ***dynamic structure***

Aspects of the Arpeggigon



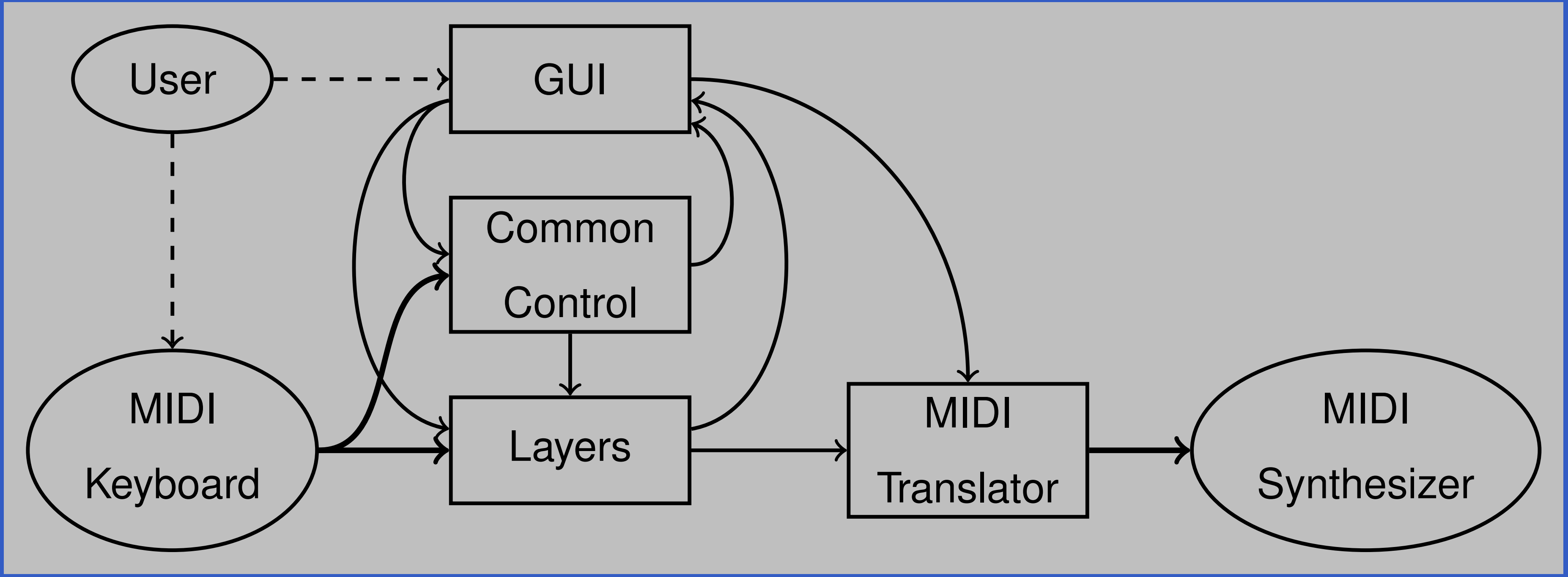
- ***Interactive***
- Layers can be added/removed: ***dynamic structure***
- Notes generated at ***discrete*** points in time

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:

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

Lifted into a signal function primarily using $accumBy$:

$$accumBy :: (b \to a \to b) \to b \to SF\ (Event\ a)\ (Event\ b)$$

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

Automated Smooth Tempo Change

Smooth transition between two preset tempos:

$$smoothTempo :: Tempo \rightarrow SF\ (Bool, Tempo, Tempo, Rate)\ Temp$$

$$\begin{array}{l}
smoothTempo\ tpo0 = \mathbf{proc}\ (sel1, tpo1, tpo2, rate) \rightarrow \mathbf{do} \\
\quad \mathbf{rec} \\
\quad\quad \mathbf{let}\ desTpo = \mathbf{if}\ sel1\ \mathbf{then}\ tpo1\ \mathbf{else}\ tpo2 \\
\quad\quad\quad\quad diff \quad = desTpo - curTpo \\
\quad\quad\quad\quad rate' \quad = \mathbf{if}\ \quad diff > 0.1\ \mathbf{then}\ rate \\
\quad\quad\quad\quad\quad\quad \mathbf{else\ if}\ diff < -0.1\ \mathbf{then}\ -rate \\
\quad\quad\quad\quad\quad\quad \mathbf{else}\ \quad 0 \\
\quad\quad curTpo \leftarrow arr\ (+tpo0) \lll integral \prec rate' \\
\quad returnA \prec curTpo
\end{array}$$

Reactive Values and Relations (1)

- The Arpeggigon interacts with the outside world using two imperative toolkits:
  - GUI: GTK+
  - MIDI I/O: Jack

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.

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.

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, . . .

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, . . .
- A Reactive Relation (RR) is a relation between RVs that is maintained automatically.

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, . . .
- A Reactive Relation (RR) is a relation between RVs that is maintained automatically.
- RVR programming takes place in the IO monad, allowing arbitrary interfacing with imperative APIs.

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, . . .
- A Reactive Relation (RR) is a relation between RVs that is maintained automatically.
- RVR programming takes place in the IO monad, allowing arbitrary interfacing with imperative APIs.
- Yet, the high-level view is quite declarative/FRP-like.

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.

Pause

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

$$
\begin{array}{l}
tempoRV' = \\
\quad liftR2\ (\lambda tempo\ paused \rightarrow \mathbf{if}\ paused\ \mathbf{then}\ 0\ \mathbf{else}\ tempo) \\
\qquad\qquad tempoRV \\
\qquad\qquad pauseButtonRV
\end{array}
$$

Pause

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

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

- This is an equation defining $tempoRV'$ 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.

Summary

Summary

- Yampa (FRP) good fit for writing interactive musical applications in a declarative way.

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.

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.

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.