The Arpeggigon: A Functional Reactive Musical Automaton

London Haskell Meetup, 2017-06-28

Henrik Nilsson

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

The Arpeggigon (2)

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

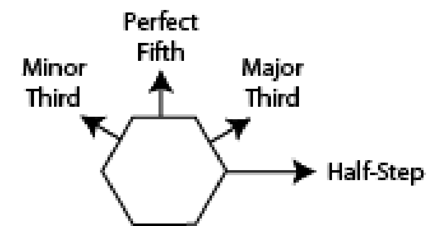
The Arpeggigon (2)

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

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

The Harmonic Table

	B		C		C#		D		D#		E		F
G		G#		A		A#		B		C		C#	
	E		F		F#		G		G#		A		A#
C		C#		D		D#		E		F		F#	
	A		A#		B		C		C#		D		D#
F		F#		G		G#		A		A#		B	
	D		D#		E		F		F#		G		G#
A#		B		C		C#		D		D#		E	
	G		G#		A		A#		B		C		C#
D#		E		F		F#		G		G#		A	
	C		C#		D		D#		E		F		F#
G#		A		A#		B		C		C#		D	
	F		F#		G		G#		A		A#		B
C#		D		D#		E		F		F#		G	
	A#		B		C		C#		D		D#		E



Running a Sample Configuration

Layer

Tempo

120

Layer beat

Quarter note

Volume 99

Strength 1.0

Repeat count

1

Beats per bar

4

Enable repeat count

Keep heads on restart

Acoustic Grand Piano

NoAccent

NoSlide

Quarter note

1

Add layer

Remove layer

Save configuration

Load configuration

Restart

Pause

Stop

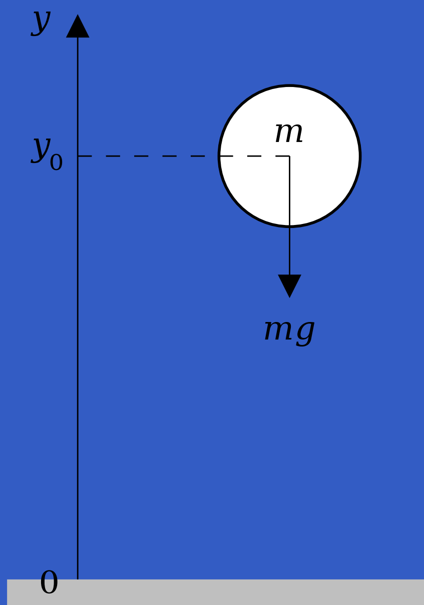
Record

This Talk

- Demonstration
- Brief introduction to FRP and Yampa
- Time in music
- 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.



$$
\begin{aligned}
y &= y_0 + \int v \, \mathrm{d}t \\
v &= v_0 + \int -9.81
\end{aligned}
$$

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

Functional Reactive Programming (2)

- FRP originated in Conal Elliott and Paul Hudak's work on Functional Reactive Animation (Fran).

Functional Reactive Programming (2)

- FRP originated in Conal Elliott and Paul Hudak's work on Functional Reactive Animation (Fran). Highly cited 1997 ICFP paper; ICFP award for most influential paper in 2007.

Functional Reactive Programming (2)

- FRP originated in Conal Elliott and Paul Hudak's work on Functional Reactive Animation (Fran). Highly cited 1997 ICFP paper; ICFP award for most influential paper in 2007.
- FRP has evolved in a number of directions and into different concrete implementations.

Functional Reactive Programming (2)

- FRP originated in Conal Elliott and Paul Hudak's work on Functional Reactive Animation (Fran). Highly cited 1997 ICFP paper; ICFP award for most influential paper in 2007.
- FRP has evolved in a number of directions and into different concrete implementations.
- We will use Yampa: an FRP system embedded in Haskell.

Key FRP Features

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

Key FRP Features

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

- Synchronous

Key FRP Features

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

Key FRP Features

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

Key FRP Features

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

Key FRP Features

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

Yampa

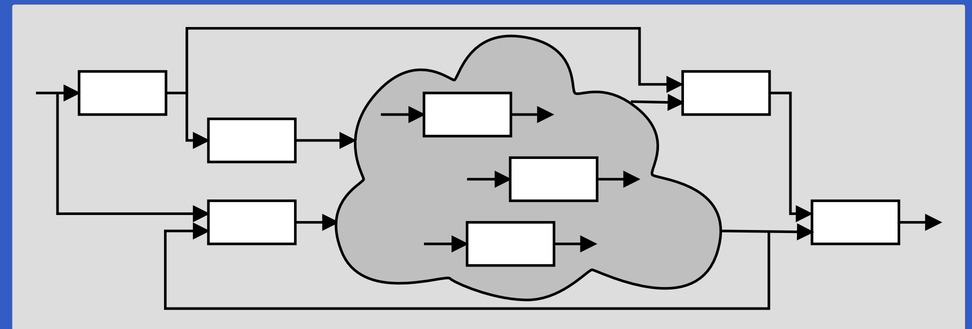
- FRP implementation embedded in Haskell

Yampa

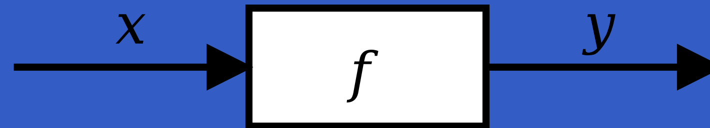
- FRP implementation embedded in Haskell
- Key notions:
  - ***Signals***: time-varying values
  - ***Signal Functions***: pure functions on signals
  - ***Switching***: temporal composition of signal functions

Yampa

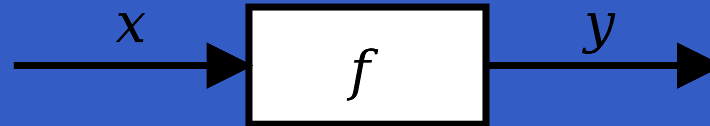
- FRP implementation embedded in Haskell
- Key notions:
  - ***Signals***: time-varying values
  - ***Signal Functions***: pure functions on signals
  - ***Switching***: temporal composition of signal functions
- Programming model:



Signal Functions

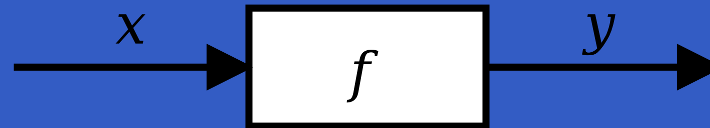


Signal Functions



Intuition:

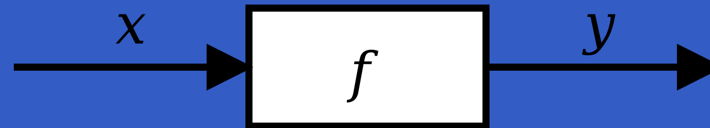
Signal Functions



Intuition:

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

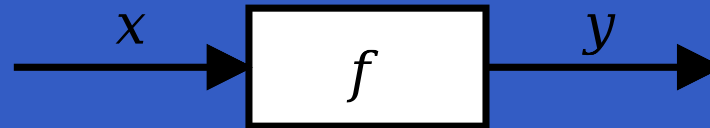
Signal Functions



Intuition:

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

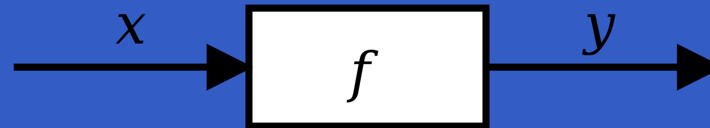
Signal Functions



Intuition:

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

Signal Functions



Intuition:

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

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

Some Basic Signal Functions

$$identity :: SF\ a\ a$$

Some Basic Signal Functions

$$identity :: SF\ a\ a$$

$$constant :: b \rightarrow SF\ a\ b$$

Some Basic Signal Functions

$$identity :: SF\ a\ a$$

$$constant :: b \rightarrow SF\ a\ b$$

$$iPre :: a \rightarrow SF\ a\ a$$

Some Basic Signal Functions

$$identity :: SF\ \ a\ \ a$$

$$constant :: b \rightarrow SF\ \ a\ b$$

$$iPre :: a \rightarrow SF\ \ a\ \ a$$

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

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

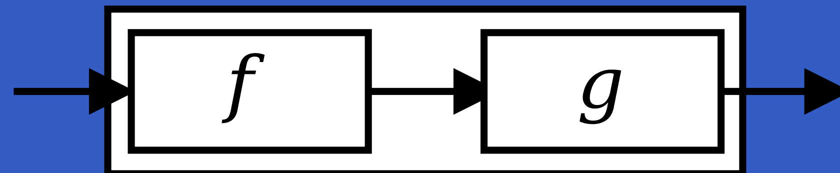
Composition

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

Composition

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

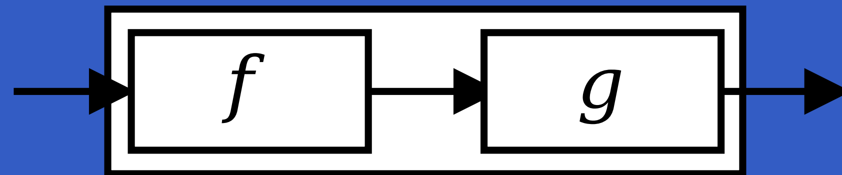
For example, serial composition:



Composition

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

For example, serial composition:



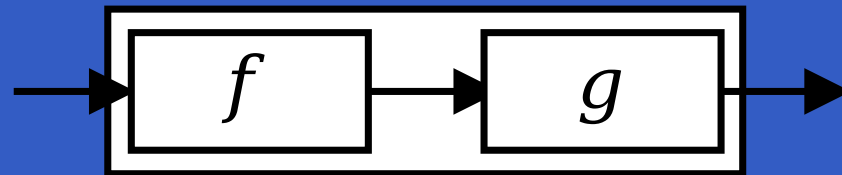
A ***combinator*** that captures this idea:

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

Composition

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

For example, serial composition:

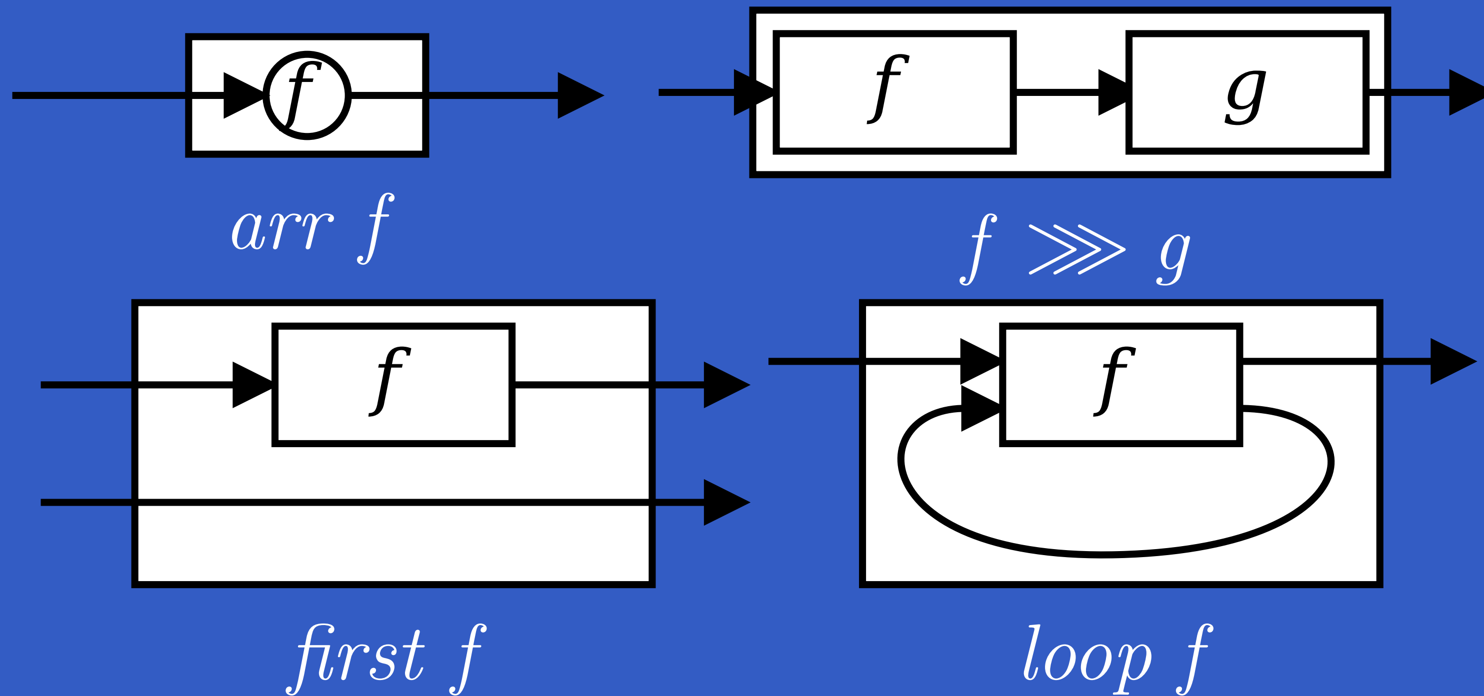


A ***combinator*** that captures this idea:

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

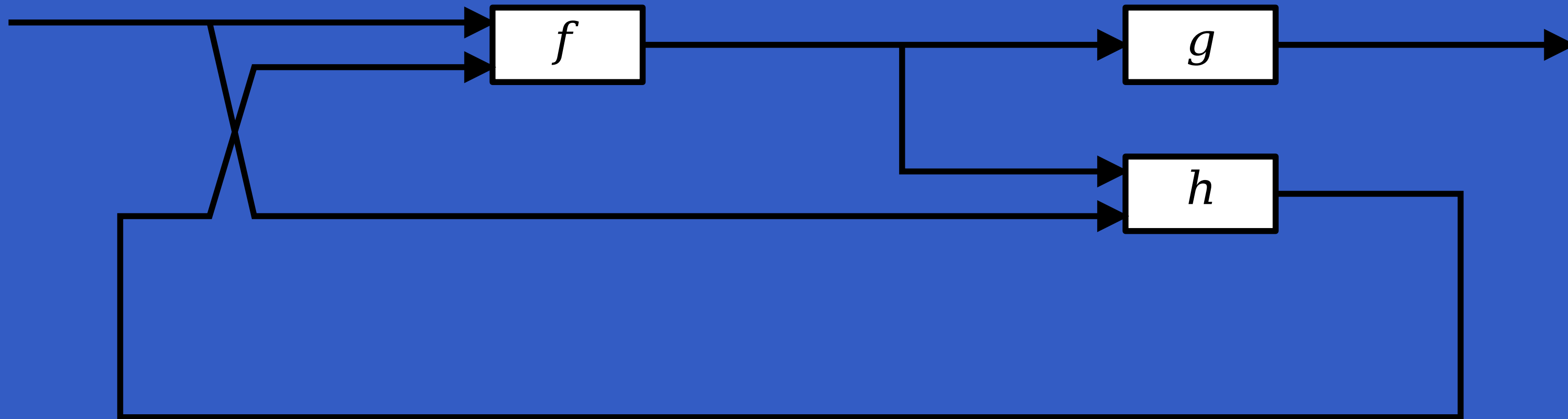
Signal functions are the primary notion; signals a secondary one, only existing indirectly.

The Arrow Combinators

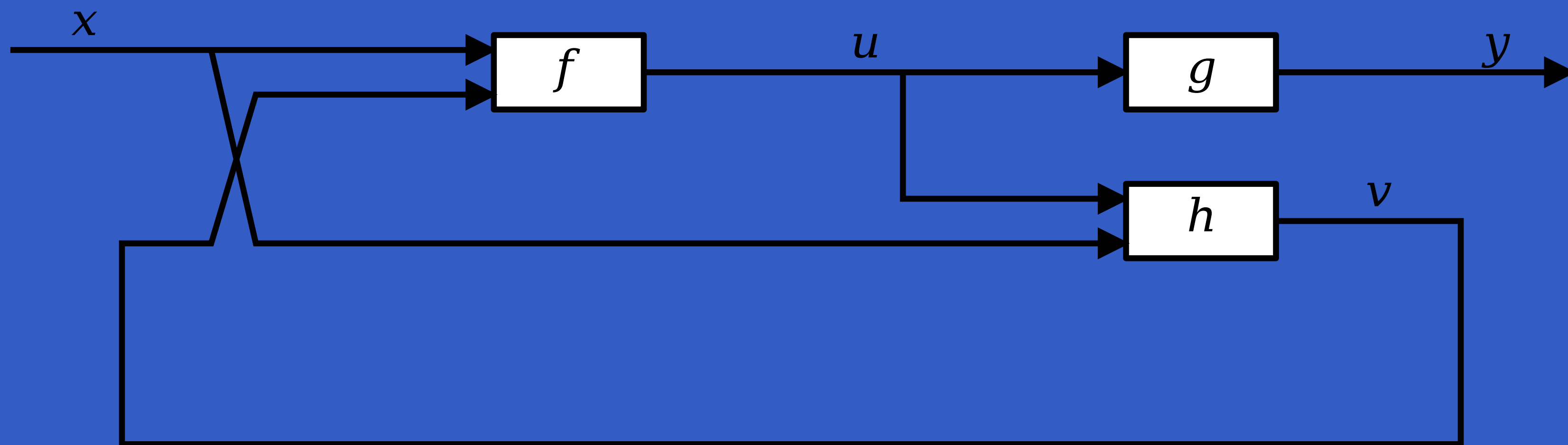


$$arr :: (a \to b) \to SF\ a\ b$$
$$(\ggg) :: SF\ a\ b \to SF\ b\ c \to SF\ a\ c$$
$$first :: SF\ a\ b \to SF\ (a, c)\ (b, c)$$
$$loop :: SF\ (a, c)\ (b, c) \to SF\ a\ b$$

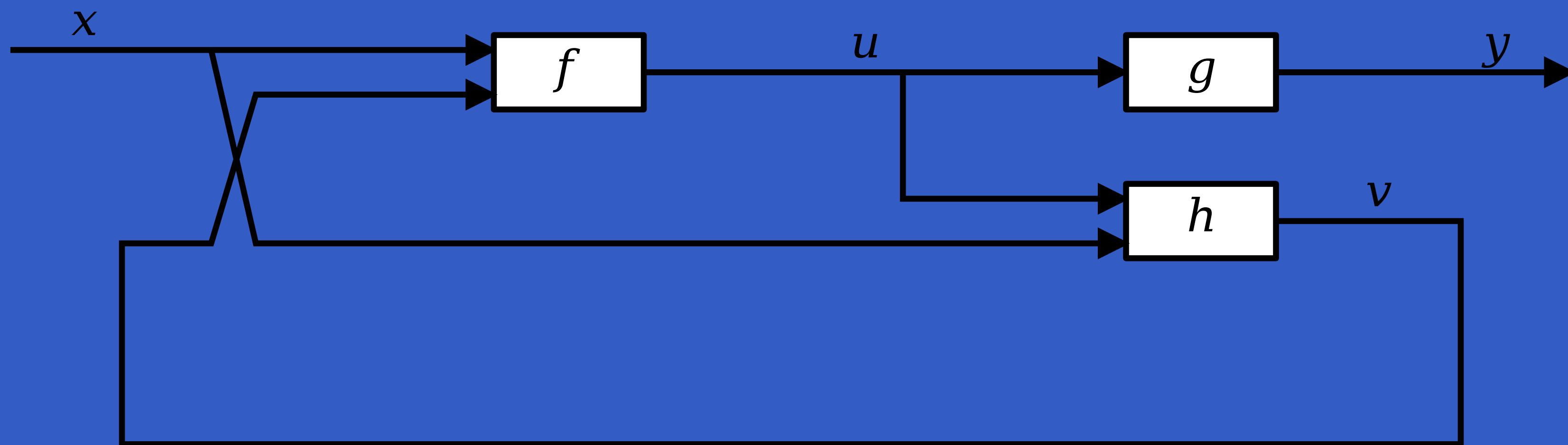
Paterson's Arrow Notation



Paterson's Arrow Notation



Paterson's Arrow Notation



proc $x \rightarrow$ **do**

rec

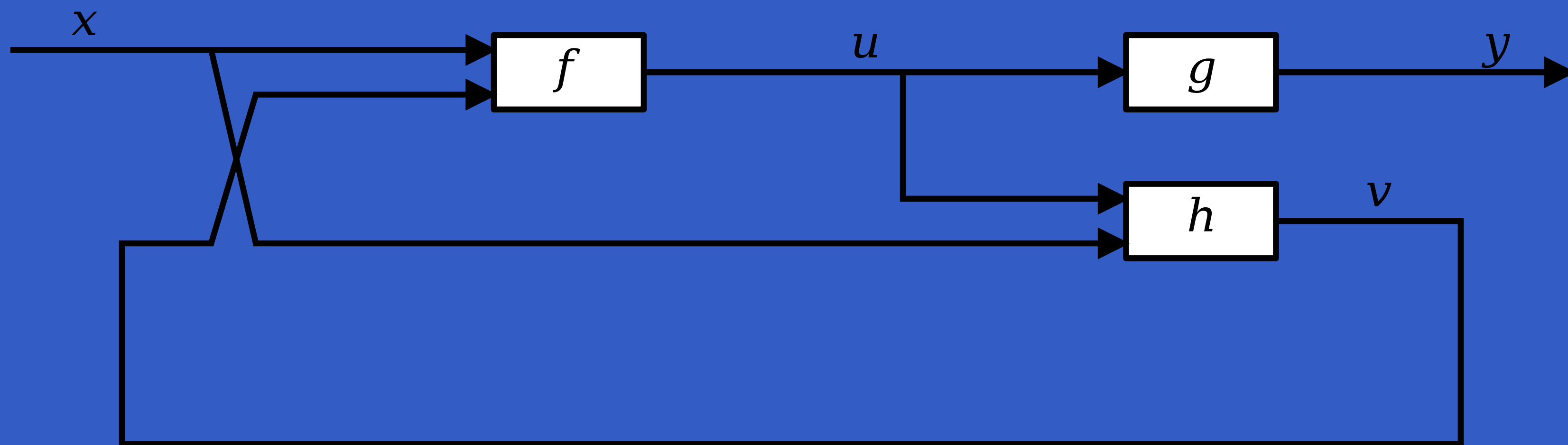
$u \leftarrow f \prec (x, v)$

$y \leftarrow g \prec u$

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

$returnA \prec y$

Paterson's Arrow Notation



proc $x \rightarrow$ **do**

rec

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

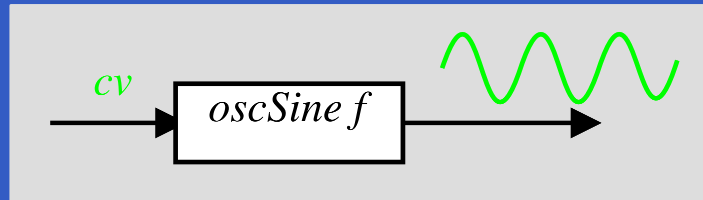
$y \leftarrow g \prec u$

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

$returnA \prec y$

Only syntactic sugar: everything translated into a combinator expression.

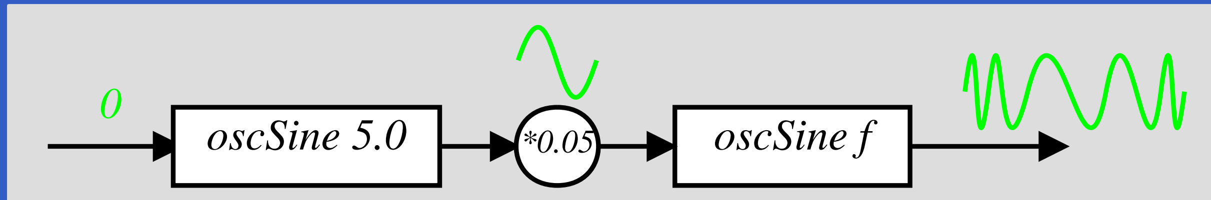
Example 1: Sine oscillator



$$oscSine :: Frequency \to SF\ \ CV\ \ Sample$$
$$oscSine\ f0 = \mathbf{proc}\ cv \to \mathbf{do}$$
$$\quad \mathbf{let}\ f = f0 * (2 ** cv)$$
$$\quad phi \leftarrow integral \prec 2 * pi * f$$
$$\quad returnA \prec sin\ phi$$

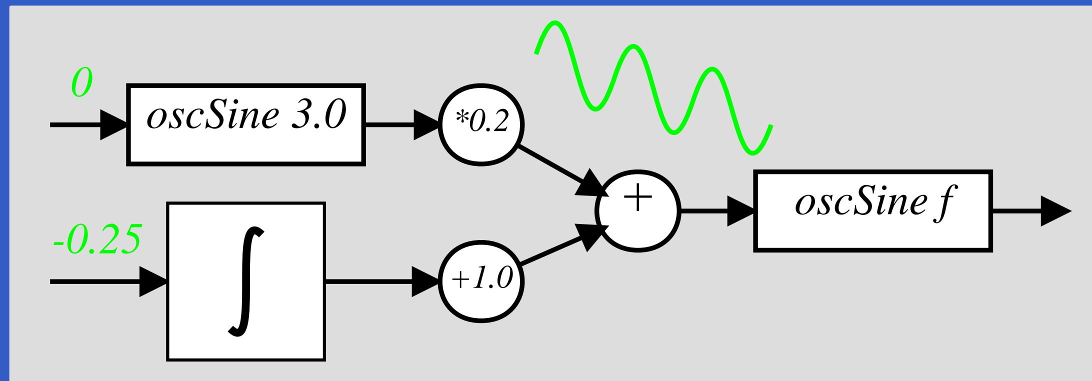
$$constant\ 0 \ggg oscSine\ 440$$

Example 2: Vibrato



$$constant\ 0$$
$$\ggg\ oscSine\ 5.0$$
$$\ggg\ arr\ (*0.05)$$
$$\ggg\ oscSine\ 440$$

Example 3: 50's Sci Fi



$$
\begin{array}{l}
sciFi :: SF\ ()\ Sample \\
sciFi = \textbf{proc}\ () \rightarrow \textbf{do} \\
\quad und \leftarrow arr\ (*0.2) \lll oscSine\ 3.0 \prec 0 \\
\quad swp \leftarrow arr\ (+1.0) \lll integral \prec -0.25 \\
\quad audio \leftarrow oscSine\ 440 \prec und + swp \\
\quad returnA \prec audio
\end{array}
$$

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` $\alpha)$.

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:

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

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

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

Switching

Q: How and when do signal functions "start"?

Switching

Q: How and when do signal functions "start"?

A:
- ***Switchers*** apply a signal functions to its input signal at some point in time.

Switching

Q: How and when do signal functions “start”?

A:
- ***Switchers*** apply a signal functions to its input signal at some point in time.
- This is ***temporal composition*** of signal functions.

Switching

Q: How and when do signal functions “start”?

A:
- ***Switchers*** apply a signal functions to its input signal at some point in time.
- This is ***temporal composition*** of signal functions.

Switchers thus allow systems with ***varying structure*** to be described.

Switching

Q: How and when do signal functions “start”?

A:
- ***Switchers*** apply a signal functions to its input signal at some point in time.
- This is ***temporal composition*** of signal functions.

Switchers thus allow systems with ***varying structure*** to be described.

Generalised switches allow composition of ***collections*** of signal functions. Can be used to model e.g. varying number of objects in a game.

The Basic Switch

Idea:

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

$$switch ::$$
$$SF\ a\ (b, Event\ c)$$
$$\rightarrow\ (c \rightarrow SF\ a\ b)$$
$$\rightarrow SF\ a\ b$$

The Basic Switch

Idea:

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

Initial SF with event source

$$switch :: SF\ a\ (b, Event\ c) \rightarrow (c \rightarrow SF\ a\ b) \rightarrow SF\ a\ b$$

The Basic Switch

Idea:

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

$$switch ::$$
$$SF\ a\ (b, Event\ c)$$
$$\rightarrow \boxed{(c \rightarrow SF\ a\ b)}$$
$$\rightarrow SF\ a\ b$$

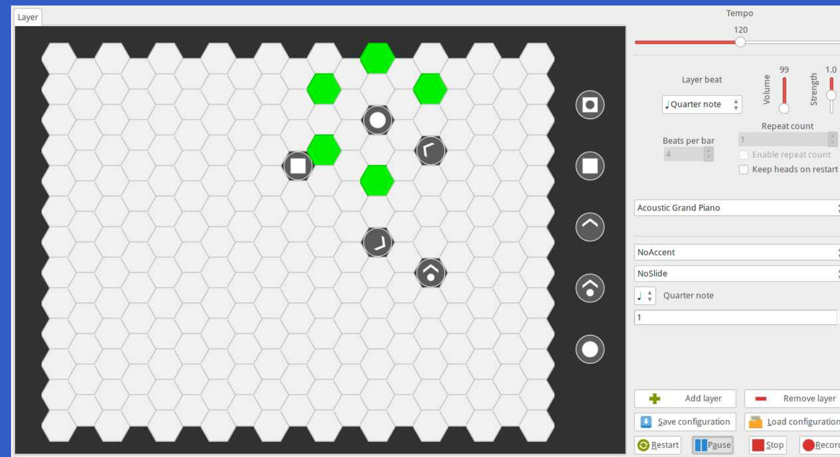
Function yielding SF to switch into

Time in Music

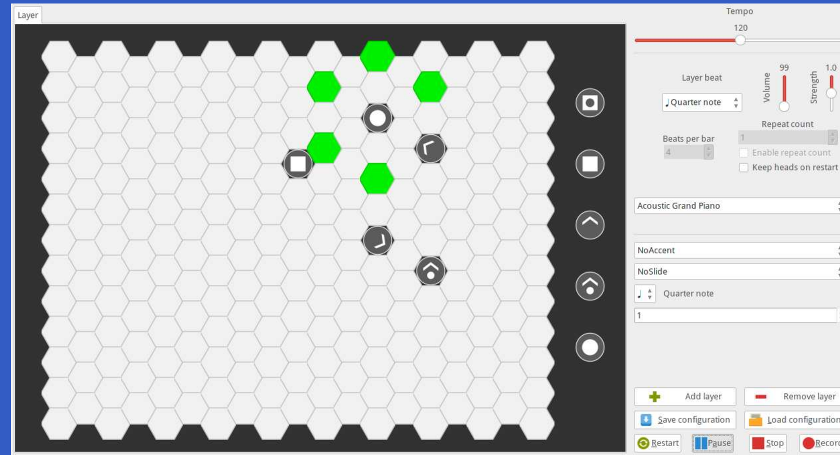
Time inherent to music. Both continuous-time and discrete-time aspects:

- Discrete or ***striated*** time:
  - Time signatures
  - Notes in a musical score
- Continuous or ***smooth*** time:
  - Crescendo
  - Ritardando
  - Portamento
  - Filter sweeps (cf. 50’s SciFi)

Aspects of the Arpeggigon (1)

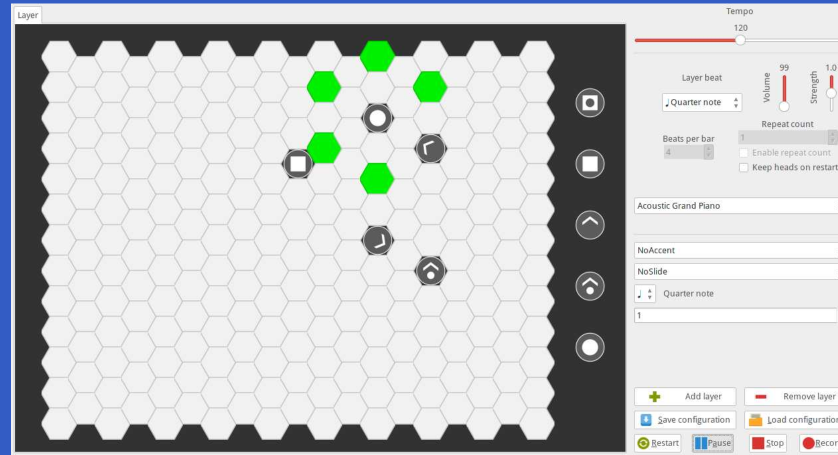


Aspects of the Arpeggigon (1)



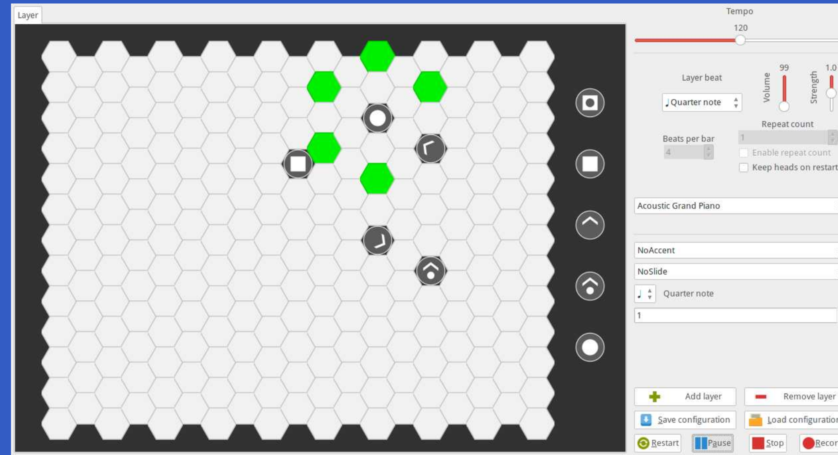
- ***Interactive***

Aspects of the Arpeggigon (1)



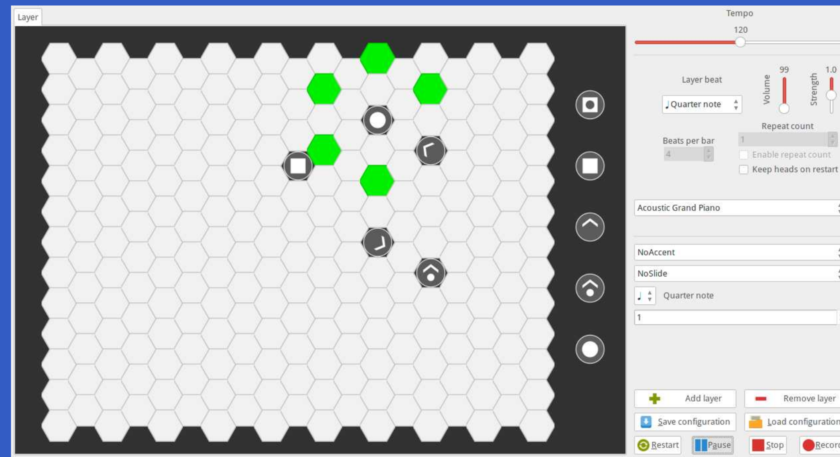
- ***Interactive***
- Layers can be added/removed: ***dynamic structure***

Aspects of the Arpeggigon (1)



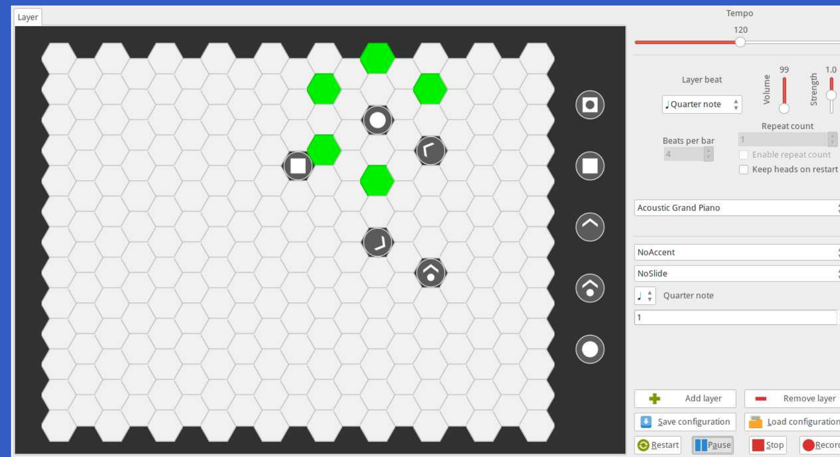
- ***Interactive***
- Layers can be added/removed: ***dynamic structure***
- Notes generated at ***discrete*** points in time

Aspects of the Arpeggigon (1)



- ***Interactive***
- Layers can be added/removed: ***dynamic structure***
- Notes generated at ***discrete*** points in time
- Notes played ***slightly shorter*** than nominal length

Aspects of the Arpeggigon (1)



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

Aspects of the Arpeggigon (2)

Potential further enhancements, e.g.:

- Swing: alternately lengthening and shortening pulse divisions
- Staccato and legato playing
- Sliding notes
- Automated, smooth, performance parameter changes

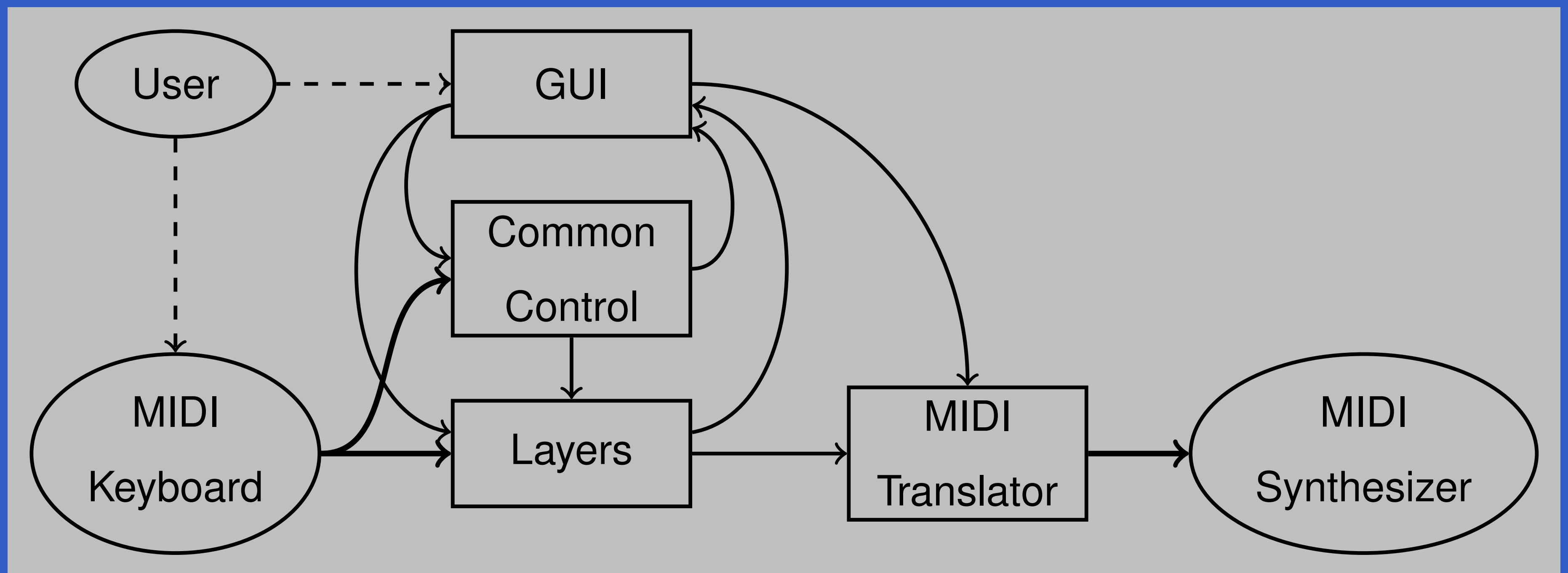
Aspects of the Arpeggigon (2)

Potential further enhancements, e.g.:

- Swing: alternately lengthening and shortening pulse divisions
- Staccato and legato playing
- Sliding notes
- Automated, smooth, performance parameter changes

Natural fit for an interactive framework supporting both discrete and continuous time. Like ***Yampa***.

Arpeggigon Architecture



Some Basic Types

$$\begin{aligned}
&\textbf{data}\ \mathit{PlayHead} = \\
&\quad \mathit{PlayHead}\ \{ \\
&\qquad \mathit{phPos} \quad :: \mathit{Pos}, \\
&\qquad \mathit{phBTM} :: \mathit{Int}, \\
&\qquad \mathit{phDir} \quad :: \mathit{Dir} \\
&\quad \}
\end{aligned}$$

$$\begin{aligned}
&\textbf{data}\ \mathit{Note} = \mathit{Note}\ \{ \\
&\quad \mathit{notePch} :: \mathit{Pitch}, \\
&\quad \mathit{noteStr} \ :: \mathit{Strength}, \\
&\quad \mathit{noteDur} :: \mathit{Duration}, \\
&\quad \mathit{noteOrn} :: \mathit{Ornaments} \\
&\}
\end{aligned}$$

Cellular Automaton

State transition function for the cellular automaton:

$$advanceHeads :: Board \to BeatNo \to RelPitch \to Strength \to [PlayHead] \to ([PlayHead], [Note])$$

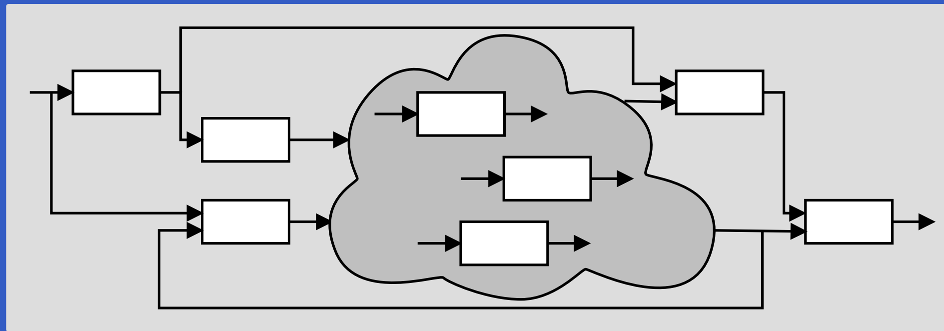
Lifted into a signal function primarily using $accumBy$:

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

$$automaton :: [PlayHead] \to SF\ (Board, DynamicLayerCtrl, Event\ BeatNo)\ (Event\ [Note], [PlayHead])$$

Layers (1)

- A layer has two states: running and stopped
- Swithing allows for:
  - Moving between states
  - Adding and removing layers dynamically



Layers (2)

A running layer is an instance of $automaton$ along with a meteronome:

$$
\begin{aligned}
&layerRunning :: StaticLayerCtrl \rightarrow [PlayHead] \\
&\quad \rightarrow SF\ (Event\ AbsBeat, Board, LayerCtrl, Event\ RunStatus) \\
&\qquad\qquad (Event\ [Note], [PlayHead]) \\
&layerRunning\ islc\ iphs = \\
&\quad switch\ (lrAux\ islc\ iphs)\ \$\ \lambda(rs', islc', iphs') \rightarrow \\
&\qquad \mathbf{case}\ rs'\ \mathbf{of} \\
&\qquad\quad Stopped\ \ \rightarrow layerStopped \\
&\qquad\quad Running \rightarrow layerRunning\ islc'\ iphs'
\end{aligned}
$$

Layers (3)

$$
\begin{aligned}
& \mathit{lrAux}\ \mathit{islc}\ \mathit{iphs} = \mathbf{proc}\ (\mathit{clk}, b, (\mathit{slc}, \mathit{dlc}, _), \mathit{ers}) \rightarrow \mathbf{do} \\
& \quad \mathit{lbc} \quad\ \ \leftarrow \mathit{layerMetronome}\ \mathit{islc} \prec (\mathit{clk}, \mathit{dlc}) \\
& \quad \mathit{enphs} \leftarrow \mathit{automaton}\ \mathit{iphs} \prec (b, \mathit{dlc}, \mathit{lbc}) \\
& \quad e \qquad\ \leftarrow \mathit{notYet} \prec \mathit{fmap}\ (\lambda \mathit{rs} \rightarrow (\mathit{rs}, \mathit{slc}, \mathit{startHeads}\ b))\ \mathit{ers} \\
& \quad \mathit{returnA} \prec (\mathit{enphs}, e)
\end{aligned}
$$

The static part of $\mathit{LayerControl}$ are parameters can't usefully be changed while the automaton is running. slc is ***sampled*** at the point of switching, and becomes the new islc. The board b is sampled as well and used to compute the new iphs.

Automatic Restarting of a Layer

A useful feature is to allow optional automatic restart of a layer every n bars.

An additional static layer parameter $restart :: Maybe\ int$ along with the following modificatio to $lrAux$ achieves this:

$$
\begin{aligned}
& r \leftarrow \textbf{case}\ restart\ islc\ \textbf{of} \\
& \qquad Nothing \rightarrow never \\
& \qquad Just\ n \quad \rightarrow countTo\ (n * barLength + 1) \\
& \quad \prec lbc \\
& \textbf{let}\ ers' = ers\ `lMerge`\ (r\ `tag`\ Running) \\
& e \leftarrow notYet \prec fmap\ (\lambda rs \rightarrow (rs, slc, startHeads\ b))\ ers'
\end{aligned}
$$

Automated Smooth Tempo Change

Smooth transition between two preset tempos:

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

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 ***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, . . .

  For example, the text field of a text input widget becomes an RV.

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.

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.

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.

Pause

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

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

Pause

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

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

- 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. After writing a value to the input, the corresponding output at that point in time can be read.

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 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.