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:

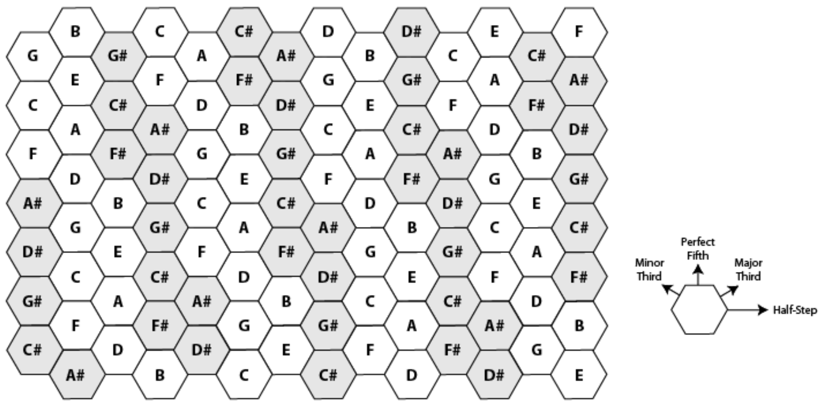
- Interactive cellular automaton:
  - Configuration
  - Performance parameters

The Arpeggigon (2)

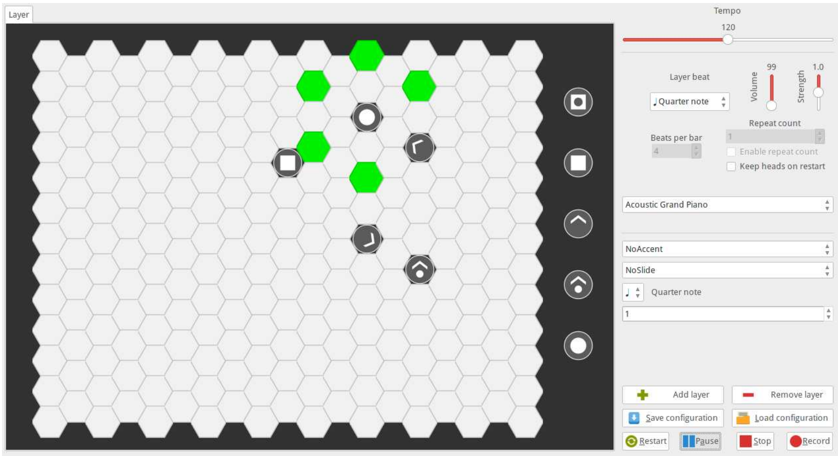
- Implemented in Haskell using:
  - The Functional Reactive Programming (FRP) system Yampa
  - Reactive Values and Relations
- Based on the *Harmonic Table*

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

The Harmonic Table



Running a Sample Configuration

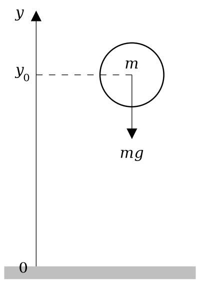


This Talk

- 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.



$$y = y_0 + \int v\,\mathrm{d}t$$

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

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

Functional Reactive Programming (2)

- 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

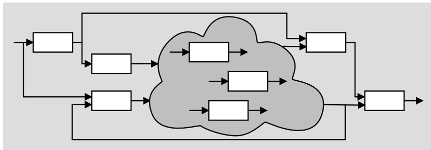
Combines conceptual simplicity of the synchronous data flow approach with the flexibility and abstraction power of higher-order functional programming:

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

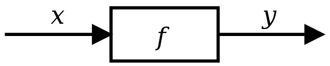
Good conceptual fit for many applications, including games and, as we will see here, interactive musical applications.

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:

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

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

Some Basic Signal Functions

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

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

$$\mathit{iPre} :: a \rightarrow \mathit{SF}\ a\ a$$

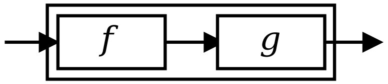
$$\mathit{integral} :: \mathit{VectorSpace}\ a\ s \Rightarrow \mathit{SF}\ a\ a$$

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

Composition

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

For example, serial composition:

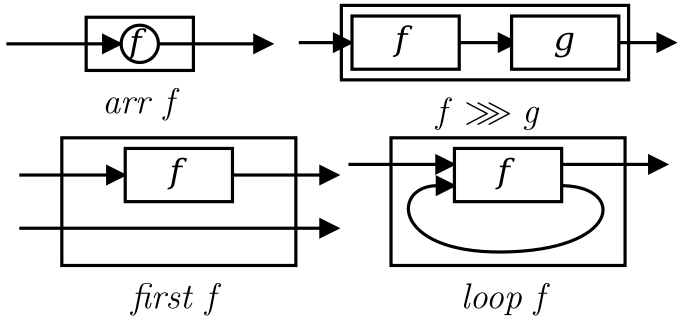


A ***combinator*** that captures this idea:

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

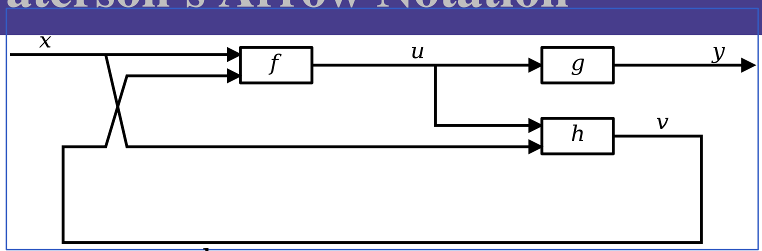
Signal functions are the primary notion; signals a secondary one, only existing indirectly.

The Arrow Combinators



$$\begin{aligned}
&arr\ \ :: (a \rightarrow b) \rightarrow SF\ a\ b\\
&(\ggg) :: SF\ a\ b \rightarrow SF\ b\ c \rightarrow SF\ a\ c\\
&first\ :: SF\ a\ b \rightarrow SF\ (a, c)\ (b, c)\\
&loop\ :: SF\ (a, c)\ (b, c) \rightarrow SF\ a\ b
\end{aligned}$$

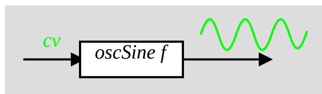
Paterson's Arrow Notation



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

Only syntactic sugar: everything translated into a combinator expression.

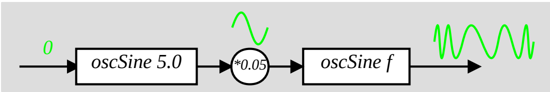
Example 1: Sine oscillator



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

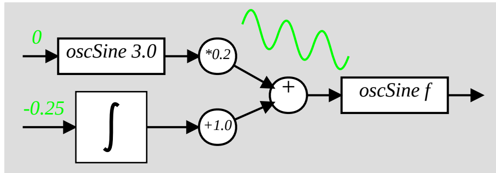
$$constant\ 0 \ggg oscSine\ 440$$

Example 2: Vibrato



$$\begin{aligned}
&constant\ 0\\
&\ggg oscSine\ 5.0\\
&\ggg arr\ (*0.05)\\
&\ggg oscSine\ 440
\end{aligned}$$

Example 3: 50's Sci Fi



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

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:

$$\begin{aligned}
&tag :: Event\ a \rightarrow b \rightarrow Event\ b\\
&after :: Time \rightarrow b \rightarrow SF\ a\ (Event\ b)\\
&edge :: SF\ Bool\ (Event\ ())
\end{aligned}$$

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.

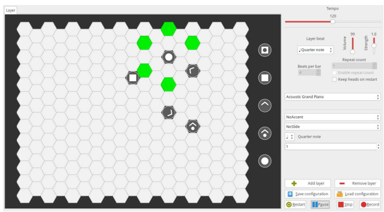
$$
\begin{aligned}
&\mathit{switch} :: \\
&\quad \mathit{SF}\ a\ (b, \mathit{Event}\ c) \\
&\quad \rightarrow (c \rightarrow \mathit{SF}\ a\ b) \\
&\quad \rightarrow \mathit{SF}\ a\ b
\end{aligned}
$$

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)



- ***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

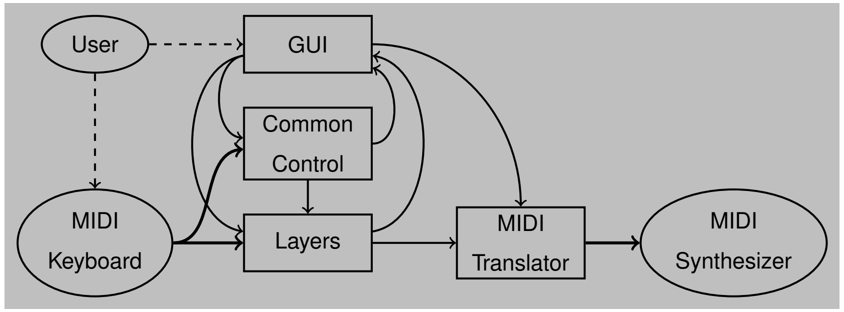
Aspects of the Arpeggigon (2)

Potential further enhancements, e.g.:

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

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

Arpeggigon Architecture



Some Basic Types

$$
\begin{array}{l}
\textbf{data } \mathit{PlayHead} = \\
\quad \mathit{PlayHead}\ \{ \\
\qquad \mathit{phPos} \quad :: \mathit{Pos}, \\
\qquad \mathit{phBTM} :: \mathit{Int}, \\
\qquad \mathit{phDir} \quad :: \mathit{Dir} \\
\quad \}
\end{array}
\qquad
\begin{array}{l}
\textbf{data } \mathit{Note} = \mathit{Note}\ \{ \\
\quad \mathit{notePch} :: \mathit{Pitch}, \\
\quad \mathit{noteStr} \ \ :: \mathit{Strength}, \\
\quad \mathit{noteDur} :: \mathit{Duration}, \\
\quad \mathit{noteOrn} :: \mathit{Ornaments} \\
\}
\end{array}
$$

Cellular Automaton

State transition function for the cellular automaton:

$$
\begin{array}{l}
\mathit{advanceHeads} :: \mathit{Board} \rightarrow \mathit{BeatNo} \rightarrow \mathit{RelPitch} \rightarrow \mathit{Strength} \\
\qquad\qquad\qquad \rightarrow [\mathit{PlayHead}] \rightarrow ([\mathit{PlayHead}], [\mathit{Note}])
\end{array}
$$

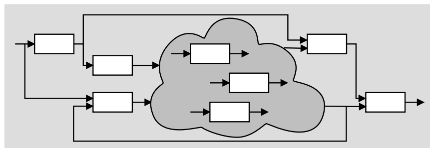
Lifted into a signal function primarily using $\mathit{accumBy}$:

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

$$
\begin{array}{l}
\mathit{automaton} :: [\mathit{PlayHead}] \\
\qquad\qquad \rightarrow \mathit{SF}\ (\mathit{Board}, \mathit{DynamicLayerCtrl}, \mathit{Event}\ \mathit{BeatNo}) \\
\qquad\qquad\qquad (\mathit{Event}\ [\mathit{Note}], [\mathit{PlayHead}])
\end{array}
$$

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 $\mathit{automaton}$ along with a meteronome:

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

Layers (3)

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

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 $\mathit{restart} :: \mathit{Maybe\ int}$ along with the following modificatio to lrAux achieves this:

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

Automated Smooth Tempo Change

Smooth transition between two preset tempos:

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

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 ***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:

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

- 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:

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

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 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.