Switched-on Yampa: Programming Modular Synthesizers in Haskell

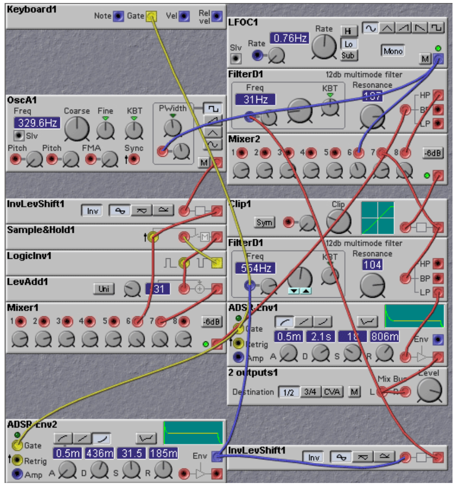
MGS Christmas Seminar 2007

Henrik Nilsson and George Giorgidze

School of Computer Science
The University of Nottingham, UK

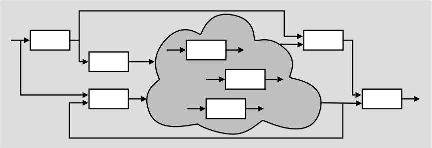
Modular synthesizers?

Modern Modular Synthesizers



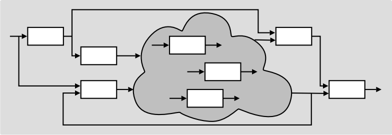
Yampa?

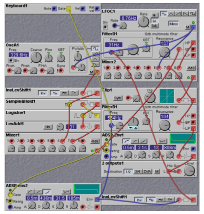
- Domain-specific language embedded in Haskell for programming *hybrid* (mixed discrete- and continuous-time) systems.
- Key concepts:
  - *Signals*: time-varying values
  - *Signal Functions*: functions on signals
  - *Switching* between signal functions
- Programming model:



What is the point?

- Music can be seen as a hybrid phenomenon. Thus interesting to explore a hybrid approach to programming music and musical applications.
- Yampa's programming model is very reminiscent of programming modular synthesizers:





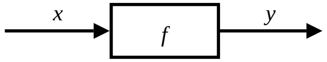
- Fun application! Useful for teaching?

What have we done?

Framework for programming modular synthesizers in Yampa:

- Sound-generating and sound-shaping modules
- Additional supporting infrastructure:
  - Input: MIDI files (musical scores), keyboard
  - Output: audio files (.wav), sound card
  - Reading SoundFont files (instrument definitions)
- Status: proof-of-concept, but decent performance.

Yampa: Signal functions



Intuition:

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

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

Yampa: Related languages

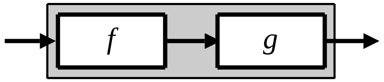
FRP/Yampa related to:

- Synchronous dataflow languages, like Esterel, Lucid Synchrone.
- Modeling languages, like Simulink, Modelica.

Yampa: Programming (1)

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

For example, serial composition:

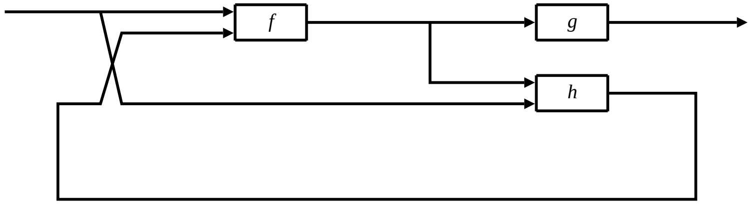


A *combinator* can be defined that captures this idea:

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

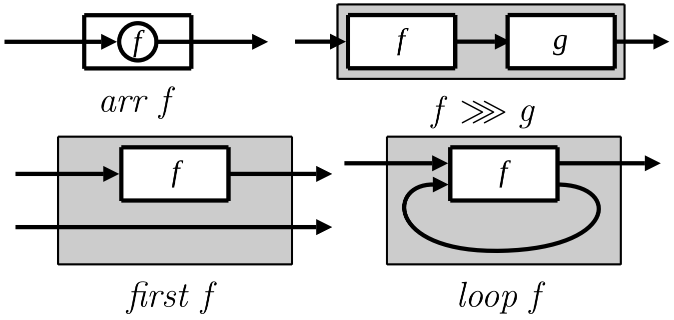
Yampa: Programming (2)

What about larger networks?
How many combinators are needed?



John Hughes's ***Arrow*** framework provides a good answer!

Yampa: The Arrow framework (1)



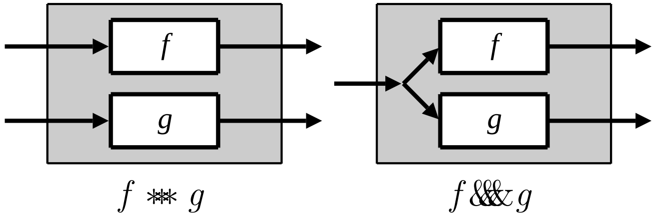
$arr\ f$ $\quad f \ggg g$

$first\ f$ $\quad loop\ f$

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

Yampa: The Arrow framework (2)

Some derived combinators:



$f *\!*\!* g$ $\quad f \&\&\& g$

$$\begin{aligned}
(*\!*\!*) &:: SF\ a\ b \rightarrow SF\ c\ d \rightarrow SF\ (a, c)\ (b, d) \\
(\&\&\&) &:: SF\ a\ b \rightarrow SF\ a\ c \rightarrow SF\ a\ (b, c)
\end{aligned}$$

Yampa: Constructing a network

Yampa: Paterson’s Arrow notation

Yampa: Discrete-time signals

Yampa’s signals are conceptually ***continuous-time*** signals.

Discrete-time signals: signals defined at discrete points in time.

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

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

Example:

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

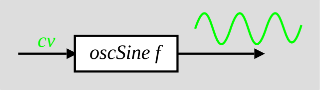
Yampa: Switching

The structure of a Yampa system may evolve over time. This is expressed through ***switching*** primitives.

Example:

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

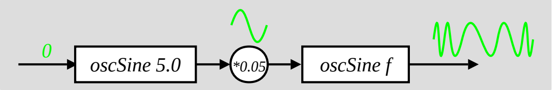
Example 1: Sine oscillator



```
oscSine :: Frequency → SF CV Sample
oscSine f0 = proc cv → do
  let f = f0 ∗ (2 ∗∗ cv)
  phi ← integral−≺ 2 ∗ pi ∗ f
  returnA−≺ sin phi
```

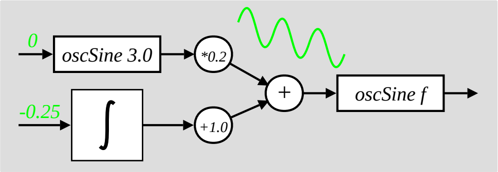
```
constant 0 ⋙ oscSine 440
```

Example 2: Vibrato



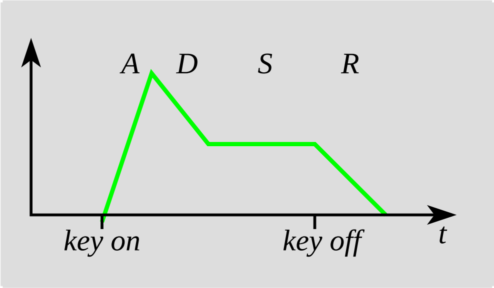
```
constant 0
⋙ oscSine 5.0
⋙ arr (∗0.05)
⋙ oscSine 440
```

Example 3: 50’s Sci Fi



```
sciFi :: SF () Sample
sciFi = proc () → do
  und   ← arr (∗0.2) ⋘ oscSine 3.0−≺ 0
  swp   ← arr (+1.0) ⋘ integral   −≺ −0.25
  audio ← oscSine 440              −≺ und + swp
  returnA−≺ audio
```

Envelope Generators (1)

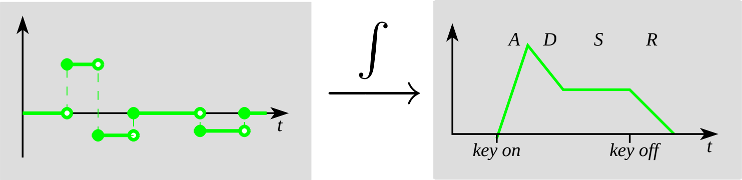


```
envGen :: CV → [(Time, CV)] → (Maybe Int)
            → SF (Event ()) (CV, Event ())
envEx = envGen 0 [(0.5, 1), (0.5, 0.5), (1.0, 0.5), (0.7, 0)]
                 (Just 3)
```

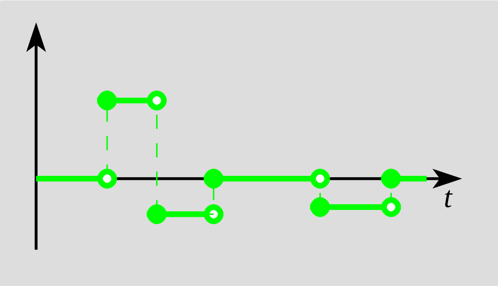
Envelope Generators (2)

How to implement?

Integration of a step function yields suitable shapes:



Envelope Generators (3)



$$
\begin{aligned}
&\mathit{afterEach} :: [(\mathit{Time}, b)] \rightarrow \mathit{SF}\ a\ (\mathit{Event}\ b)\\
&\mathit{hold} \quad\ \ :: a \rightarrow \mathit{SF}\ (\mathit{Event}\ a)\ a\\
&\mathit{steps} = \mathit{afterEach}\ [(0.7, 2), (0.5, -1), (0.5, 0), (1, -0.7), (0.7, 0)]\\
&\qquad\quad \ggg \mathit{hold}\ 0
\end{aligned}
$$

Envelope Generators (4)

Envelope generator with predetermined shape:

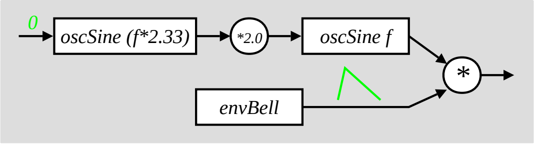
$$
\begin{aligned}
&\mathit{envGenAux} :: \mathit{CV} \rightarrow [(\mathit{Time}, \mathit{CV})] \rightarrow \mathit{SF}\ a\ \mathit{CV}\\
&\mathit{envGenAux}\ \mathit{l0}\ \mathit{tls} = \mathit{afterEach}\ \mathit{trs} \ggg \mathit{hold}\ \mathit{r0}\\
&\qquad\qquad\qquad\qquad \ggg \mathit{integral} \ggg \mathit{arr}\ (+\mathit{l0})\\
&\quad \mathbf{where}\\
&\qquad (\mathit{r0}, \mathit{trs}) = \mathit{toRates}\ \mathit{l0}\ \mathit{tls}
\end{aligned}
$$

Envelope Generators (5)

Envelope generator responding to key off:

$$
\begin{aligned}
&\mathit{envGen} :: \mathit{CV} \rightarrow [(\mathit{Time}, \mathit{CV})] \rightarrow (\mathit{Maybe}\ \mathit{Int})\\
&\qquad\quad \rightarrow \mathit{SF}\ (\mathit{Event}\ ())\ (\mathit{CV}, \mathit{Event}\ ())\\
&\mathit{envGen}\ \mathit{l0}\ \mathit{tls}\ (\mathit{Just}\ n) =\\
&\quad \mathit{switch}\ (\mathbf{proc}\ \mathit{noteoff} \rightarrow \mathbf{do}\\
&\qquad\qquad l \leftarrow \mathit{envGenAux}\ \mathit{l0}\ \mathit{tls1} \prec ()\\
&\qquad\qquad \mathit{returnA} \prec ((l, \mathit{noEvent}), \mathit{noteoff}\ \text{`}\mathit{tag}\text{`}\ l))\\
&\qquad\quad (\lambda l \rightarrow \mathit{envGenAux}\ l\ \mathit{tls2}\\
&\qquad\qquad \&\&\&\ \mathit{after}\ (\mathit{sum}\ (\mathit{map}\ \mathit{fst}\ \mathit{tls2}))\ ())\\
&\quad \mathbf{where}\\
&\qquad (\mathit{tls1}, \mathit{tls2}) = \mathit{splitAt}\ n\ \mathit{tls}
\end{aligned}
$$

Example 4: Bell



```
bell :: Frequency → SF () (Sample, Event)
bell f = proc () → do
  m           ← oscSine (2.33 ∗ f)−≺ 0
  audio       ← oscSine f         −≺ 2.0 ∗ m
  (ampl, end) ← envBell           −≺ noEvent
  returnA−≺ (audio ∗ ampl, end)
```

Example 5: Tinkling Bell

```
tinkle :: SF () Sample
tinkle = (repeatedly 0.25 84
            ⋙ constant ()
               &&& arr (fmap (bell ∘ midiNoteToFreq))
            ⋙ rSwitch (constant 0))
```

Example 6: Playing a C-major scale

```
scale :: SF () Sample
scale = (afterEach [(0.0, 60), (2.0, 62), (2.0, 64),
                    (2.0, 65), (2.0, 67), (2.0, 69),
                    (2.0, 71), (2.0, 72)]
           ⋙ constant ()
              &&& arr (fmap (bell ∘ midiNoteToFreq))
           ⋙ rSwitch (constant 0))
         &&& after 16 ()
```

Example 7: Playing simultaneous notes

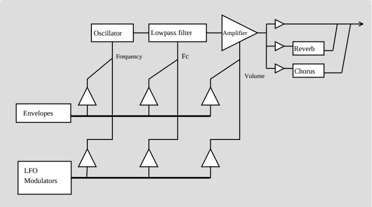
```
mysterySong :: SF () (Sample, Event ())
mysterySong = proc _ → do
  t ← tinkle   −≺ ()
  m ← mystery−≺ ()
  returnA−≺ (0.4 ∗ t + 0.6 ∗ m)
```

A polyphonic synthesizer (1)

Sample-playing monophnic synthesizer:

- Read samples (instrument recordings) from SoundFont file into internal table.
- Oscillator similar to sine oscillator, except sine func. replaced by table lookup and interpolation.

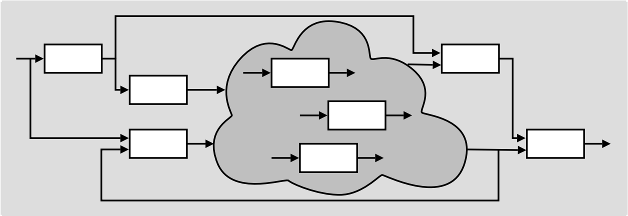
SoundFont synthesizer structure:



A polyphonic synthesizer (2)

Exploit Yampa’s switching capabilities to:

- create and switch in a mono synth instance is response to each note on event;
- switch out the instance in response to a corresponding note off event.



Switched-on Yampa?

Software and paper: `www.cs.nott.ac.uk/~ggg`