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?
Modular synthesizers?
Modular synthesizers?
Yampa?
Yampa?

- Domain-specific language embedded in Haskell for programming *hybrid* (mixed discrete- and continuous-time) systems.
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
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?
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.
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:
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:

![Diagram of Yampa's programming model]

- Fun application! Useful for teaching?
What have we done?
What have we done?

Framework for programming modular synthesizers in Yampa:
What have we done?

Framework for programming modular synthesizers in Yampa:

- Sound-generating and sound-shaping modules
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)
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

\[ x \overset{f}{\rightarrow} y \]
Yampa: Signal functions

Intuition:
Yampa: Signal functions

Intuition:

\[ \text{Time} \approx \mathbb{R} \]
Yampa: Signal functions

Intuition:

\[ \text{Time} \approx \mathbb{R} \]
\[ \text{Signal} \ a \approx \text{Time} \rightarrow a \]
\[ x :: \text{Signal} \ T1 \]
\[ y :: \text{Signal} \ T2 \]
Yampa: Signal functions

Intuition:

\[ Time \approx \mathbb{R} \]
\[ Signal \ a \approx Time \rightarrow a \]
\[ x :: Signal \ T1 \]
\[ y :: Signal \ T2 \]
\[ SF \ a \ b \approx Signal \ a \rightarrow Signal \ b \]
\[ f :: SF \ T1 \ T2 \]
Yampa: 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, \textit{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.
In Yampa, systems are described by combining signal functions (forming new signal functions).
In Yampa, systems are described by combining signal functions (forming new signal functions). For example, serial composition:
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:

\[
(\ggg) :: SF\ a\ b \rightarrow SF\ b\ c \rightarrow SF\ a\ c
\]
What about larger networks? How many combinators are needed?
What about larger networks? How many combinators are needed?

John Hughes’s *Arrow* framework provides a good answer!
Yampa: The Arrow framework (1)

\[ \text{arr } f \]

\[ \text{first } f \]

\[ \text{loop } f \]

\[
\text{arr} \quad :: \quad (a \rightarrow b) \rightarrow SF \ a \ b \\
(\ggg) \quad :: \quad SF \ a \ b \rightarrow SF \ b \ c \rightarrow SF \ a \ c \\
\text{first} \quad :: \quad SF \ a \ b \rightarrow SF \ (a, c) \ (b, c) \\
\text{loop} \quad :: \quad SF \ (a, c) \ (b, c) \rightarrow SF \ a \ b
\]
Some derived combinators:

\[ f \starstar g \]

\[ f \&\& g \]

\[(\starstar) :: SF \ a \ b \rightarrow SF \ c \ d \rightarrow SF \ (a, c) \ (b, d)\]

\[(\&\&): SF \ a \ b \rightarrow SF \ a \ c \rightarrow SF \ a \ (b, c)\]
Yampa: Constructing a network
Yampa: Constructing a network
Yampa: Constructing a network

\[
\text{loop } (\text{arr } (\lambda(x, y) \rightarrow ((x, y), x))) \\
\ggg (\text{first } f) \\
\ggg (\text{arr } (\lambda(x, y) \rightarrow (x, (x, y)))) \ggg (g \star h))
\]
Yampa: Paterson’s Arrow notation
Yampa: Paterson’s Arrow notation
proc $x \rightarrow$ do

rec

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

$y \leftarrow g \prec u$

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

$returnA \prec y$
Yampa: Discrete-time signals

Yampa’s signals are conceptually continuous-time signals.
Yampa: Discrete-time signals

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

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

\[
\text{data } \text{Event } a = \text{NoEvent} \mid \text{Event } a
\]

Example:

\[
\text{repeatedly :: Time } \to b \to SF a \ (\text{Event } b)
\]
Yampa: Switching

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

Example:

\[
\text{switch} :: SF \ a \ (b, \ Event \ c) \to (c \to SF \ a \ b) \\
\to SF \ a \ b
\]
Example 1: Sine oscillator

\[ \text{oscSine} :: \text{Frequency} \rightarrow \text{SF CV Sample} \]

\[ \text{oscSine} \ f0 = \text{proc} \ cv \rightarrow \text{do} \]

\[ \quad \text{let} \ f = f0 \ast (2 \ast \ast \ cv) \]

\[ \quad \text{phi} \leftarrow \text{integral} \left(2 \ast \pi \ast f\right) \]

\[ \quad \text{returnA} \leftarrow \sin \phi \]

\[ \text{constant} \ 0 \ \gg \ \text{oscSine} \ 440 \]
Example 2: Vibrato

constant 0
⇒ oscSine 5.0
⇒ arr (*0.05)
⇒ oscSine 440
Example 3: 50’s Sci Fi

\[osciSine 3.0 \times 0.2 \rightarrow oscSine f\]

\[sciFi \:: SF () \rightarrow Sample\]

\[sciFi = proc () \rightarrow do\]

\[und \leftarrow arr (*0.2) \ll oscSine 3.0 \leftarrow 0\]

\[swp \leftarrow arr (+1.0) \ll integral \leftarrow -0.25\]

\[audio \leftarrow oscSine 440 \leftarrow und + swp\]

\[returnA \leftarrow audio\]
Envelope Generators (1)

\[ \text{envGen} :: CV \rightarrow [(\text{Time}, CV)] \rightarrow (\text{Maybe Int}) \]
\[ \rightarrow SF (\text{Event} ()) (CV, \text{Event} ()) \]

\[ \text{envEx} = \text{envGen} 0 [(0.5, 1), (0.5, 0.5), (1.0, 0.5), (0.7, 0)] \]
\[ (\text{Just} 3) \]
Envelope Generators (2)

How to implement?
Envelope Generators (2)

How to implement?

Integration of a step function yields suitable shapes:
Envelope Generators (3)

\[
\text{afterEach} :: [(\text{Time}, b)] \rightarrow \text{SF} \ a \ (\text{Event} \ b)
\]

\[
\text{hold} :: a \rightarrow \text{SF} \ (\text{Event} \ a) \ a
\]

\[
\text{steps} = \text{afterEach} \ [(0.7, 2), (0.5, -1), (0.5, 0), (1, -0.7), (0.7, 0)]
\]

\[\Rightarrow \text{hold} \ 0\]
Envelope Generators (4)

Envelope generator with predetermined shape:

\[ envGenAux :: CV \rightarrow [(Time, CV)] \rightarrow SF a CV \]

\[ envGenAux l0 tls = \text{afterEach} \ trs \gg \text{hold} \ r0 \gg \text{integral} \gg \text{arr} \ (+l0) \]

where

\[ (r0, trs) = \text{toRates} \ l0 \ tls \]
Envelopes Generators (5)

Envelope generator responding to key off:

\[
\text{envGen} :: CV \to [(\mathit{Time}, CV)] \to (\text{Maybe Int}) \\
\to SF (\mathit{Event} ()) (CV, \mathit{Event} ())[143]
\]

\[
\text{envGen} l0 \; \text{tls} \; (\text{Just} \; n) =
\]

switch (\textbf{proc} \; \text{noteoff} \rightarrow \textbf{do} \\
\quad l \leftarrow \text{envGenAux} \; l0 \; \text{tls1} \\n\quad \text{return}A \leftarrow ((l, \text{noEvent}), \text{noteoff} \; \text{‘tag’} \; l) \\
\quad (\lambda l \rightarrow \text{envGenAux} \; l \; \text{tls2} \\
\quad & & \text{\&\& after} \; (\text{sum} \; (\text{map} \; \text{fst} \; \text{tls2})) \; ()))
\]

where

\[
(tls1, tls2) = \text{splitAt} \; n \; \text{tls}
\]
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

\[
tinkle :: SF () \text{ Sample} \\
tinkle = (\text{repeatedly } 0.25 \ 84) \\
\quad \gg \quad \text{constant } () \\
\quad \gg \quad \text{arr } (\text{fmap } (\text{bell } \circ \text{midiNoteToFreq})) \\
\quad \gg \quad \text{rSwitch } (\text{constant } 0))
\]
Example 6: Playing a C-major scale

\[
\text{scale :: SF () Sample}
\]

\[
\text{scale = (afterEach [(0.0, 60), (2.0, 62), (2.0, 64),}
\]

\[
\quad (2.0, 65), (2.0, 67), (2.0, 69),
\]

\[
\quad (2.0, 71), (2.0, 72)]
\]

\[
\ggg \text{constant ()}
\]

\[
\&\& arr (fmap (bell \circ \text{midiNoteToFreq}))
\]

\[
\ggg rSwitch (\text{constant 0})
\]

\[
\&\& after 16 ()
\]
Example 7: Playing simultaneous notes

\[
\text{mysterySong :: SF} \ (\text{Sample, Event} \ () ) \\
mysterySong = \text{proc } \_ \rightarrow \text{do} \\
\hspace{1cm} \text{\quad } t \leftarrow \text{tinkle} \quad \leftarrow () \\
\hspace{1cm} \text{\quad } m \leftarrow \text{mystery} \leftarrow () \\
\hspace{1cm} \text{\quad return} \ A \leftarrow (0.4 \ast t + 0.6 \ast m) 
\]
A polyphonic synthesizer (1)

Sample-playing monophonic 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:
Exploit Yampa’s switching capabilities to:

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

Software and paper: [www.cs.nott.ac.uk/~ggg](http://www.cs.nott.ac.uk/~ggg)