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:

\[
\begin{align*}
\text{Time} & \approx \mathbb{R} \\
\text{Signal} a & \approx \text{Time} \to a \\
x & :: \text{Signal} \ T1 \\
y & :: \text{Signal} \ T2 \\
\text{SF} \ a \ b & \approx \text{Signal} \ a \to \text{Signal} \ b \\
f & :: \text{SF} \ T1 \ T2
\end{align*}
\]

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:

```
\[ f \gg g \rightarrow SF a \rightarrow SF b \rightarrow SF c \rightarrow SF a c \]
```

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

```
(\gg) :: SF a b \rightarrow SF b c \rightarrow SF a c
```

**Yampa: The Arrow framework (1)**

```
arr f :: (a \rightarrow) \rightarrow SF a b
(\gg) :: SF a b \rightarrow SF b c \rightarrow SF a c
first :: SF a b \rightarrow SF (a, c) \rightarrow SF (b, c)
loop :: SF (a, c) \rightarrow SF a b
```

**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 (2)**

```
\begin{align*}
& f \\
\rightarrow & g \\
\rightarrow & h
\end{align*}
```

Some derived combinators:

```
\begin{align*}
& f \gg g \\
& f \&& g
\end{align*}
```

```
\begin{align*}
& (\gg) :: SF a b \rightarrow SF c d \rightarrow SF (a, c) \rightarrow SF (b, d) \\
& (\&&) :: SF a b \rightarrow SF a c \rightarrow SF a (b, c)
\end{align*}
```
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:

```haskell
data Event a = NoEvent | Event a
```

Example:

```haskell
repeatedly :: Time → b → SF a (Event b)
```

Yampa: Switching

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

Example:

```haskell
switch :: SF a (b, Event c) → (c → SF a b) → SF a b
```
Example 1: Sine oscillator

oscSine :: Frequency \rightarrow SF CV Sample
oscSine f0 = proc cv \rightarrow do
  let f = f0 \times (2 ** cv)
  phi \leftarrow integral \rightarrow 2 \times pi \times f
  returnA \leftarrow \sin \ phi

constant 0 \Rightarrow oscSine 440

Example 2: Vibrato

cv oscSine f
constant 0 \Rightarrow oscSine 5.0
\Rightarrow arr (*0.05)
\Rightarrow oscSine 440

Example 3: 50’s Sci Fi

sciFi :: SF () Sample
sciFi = proc () \rightarrow do
  und \leftarrow arr (*0.2) \ll\ll oscSine 3.0 \rightarrow 0
  swp \leftarrow arr (+1.0) \ll\ll integral \rightarrow -0.25
  audio \leftarrow oscSine 440
  returnA \leftarrow und + swp

Example 4: Envelope Generators (1)

envGen :: CV \rightarrow [(Time, CV)] \rightarrow (Maybe Int)
  \rightarrow 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:

![Graph showing integration of step function]

**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:

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

\[
\text{envGenAux} l0 \ tls = \text{afterEach} \ t r0\] \[\Rightarrow \text{integral} \Rightarrow \text{arr} (+l0) \]

\[\text{where} \]

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

**Envelope Generators (5)**

Envelope generator responding to key off:

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

\[
\text{envGen} \ l0 \ \tls \ (\text{Just} \ n) = \text{switch} \ (\text{proc} \ \text{noteoff} \rightarrow \text{do}) \]

\[
\text{where} \]

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

Programming Modular Synthesizers in Haskell – p.21/31
Programming Modular Synthesizers in Haskell – p.22/31
Programming Modular Synthesizers in Haskell – p.23/31
Programming Modular Synthesizers in Haskell – p.24/31
Example 4: Bell

\[
\text{bell} :: \text{Frequency} \rightarrow \text{SF} () \ (\text{Sample, Event})
\]
\[
bell \ f = \text{proc} () \rightarrow \text{do}
\]
\[
0 \leftarrow \text{oscSine} (2.33 * f) \leftarrow 0
\]
\[
2.0 \leftarrow \text{oscSine} f \leftarrow 2.0 * m
\]
\[
(\text{ampl, end}) \leftarrow \text{envBell} \leftarrow \text{noEvent}
\]
\[
\text{returnA} \leftarrow (\text{audio} * \text{ampl, end})
\]
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:

A polyphonic synthesizer (2)

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?

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