Switched-On Yampa*

Declarative Programming of Modular Synthesizers

George Giorgidze[1] and Henrik Nilsson[2]

[1] School of Computer Science, University of Nottingham, UK
ggg@cs.nott.ac.uk
[2] School of Computer Science, University of Nottingham, UK
nhn@cs.nott.ac.uk

Abstract. In this paper, we present an implementation of a modular synthesizer in Haskell using Yampa. A synthesizer, be it a hardware instrument or a pure software implementation, as here, is said to be *modular* if it provides sound-generating and sound-shaping components that can be interconnected in arbitrary ways. Yampa, a Haskell-embedded implementation of Functional Reactive Programming, supports flexible, purely declarative construction of hybrid systems. Since music is a hybrid continuous-time and discrete-time phenomenon, Yampa is a good fit for such applications, offering some unique possibilities compared to most languages targeting music or audio applications. Through the presentation of our synthesizer application, we demonstrate this point and provide insight into the Yampa approach to programming reactive, hybrid systems. We develop the synthesizer gradually, starting with fundamental synthesizer components and ending with an application that is capable of rendering a standard MIDI file as audio with respectable performance.

Keywords: Functional Reactive Programming, synchronous dataflow languages, hybrid systems, computer music.

1 Introduction

A dynamic system or phenomenon is *hybrid* if it exhibits both continuous-time and discrete-time behaviour at the chosen level of abstraction. Music is an interesting example of a hybrid phenomenon in this sense. At a fundamental level, music is sound: continuous pressure waves in some medium such as air. In contrast, a musical performance has some clear discrete aspects (along with continuous ones): it consists of sequences of discrete notes, different instruments may be played at different points of a performance, and so on.

There exist many languages and notations for describing sound or music and for programming computers to carry out musical tasks. However, they mostly tend to focus on either the discrete or the continuous aspects. Traditional musical notation, or its modern-day electronic derivatives such as MIDI files or

* This work is supported by EPSRC grant EP/D064554/1. Thanks to the anonymous referees for many useful suggestions.

P. Hudak and D.S. Warren (Eds.): PADL 2008, LNCS 4902, pp. 282–298, 2008.

domain-specific languages like Haskore [5], focus on describing music in terms of sequences of notes. If we are interested in describing music at a finer level of detail, in particular, what it actually sounds like, options include modelling languages for describing the physics of acoustic instruments, various kinds of electronic synthesizers, or domain-specific languages like Csound [14]. However, the focus of synthesizer programming is the sound of a single note, and how that sound evolves over time. The mapping between the discrete world of notes and the continuous world of sound is hard-wired into the synthesizer, outside the control of the programmer.

Here we take a more holistic approach allowing the description of both the continuous and discrete aspects of music and musical applications; that is, an approach supporting programming of *hybrid* systems. Yampa [4,10], an instance of Functional Reactive Programming (FRP) in the form of a domain-specific language embedded in Haskell, provides the prerequisite facilities. Our basic approach is that of *modular synthesis*. Modular synthesizers were developed in the late 1950s and early 1960s and offered the first programmatic way to describe sound. This was achieved by wiring together sound-generating and sound-shaping *modules* electrically. Yampa's continuous-time aspects serve this purpose very well. Additionally we leverage Yampa's capabilities for describing systems with a highly dynamic structure, thus catering for the discrete aspects of music. In this paper, we illustrate:

- how basic sound-generating and sound-shaping modules can be described and combined into a simple monophonic (one note at a time) synthesizer;
- how a monophonic synthesizer can be constructed from an instrument description contained in a SoundFont file;
- how to run several monophonic synthesizer instances simultaneously, thus creating a polyphonic synthesizer capable of playing Standard MIDI Files.

The resulting application renders the musical score in a given MIDI file using SoundFont instrument descriptions. The performance is fairly good: a moderately complex score can be rendered about as fast as it plays (with buffering). All code is available on-line.[1] In addition to what is described in this paper, the code includes supporting infrastructure for reading MIDI files, for reading SoundFont files, and for writing the result as audio files or streams (.wav).

The contribution of this work lies in the application of declarative hybrid programming to a novel application area, and as an example of advanced declarative hybrid programming. We believe it will be of interest to people interested in a declarative approach to describing music and programming musical applications, to practitioners interested in advanced declarative hybrid programming, and to educationalists seeking interesting and fun examples of declarative programming off the beaten path. The importance of the latter is illustrated by the DrScheme experience, where first-class images and appropriate reactive abstractions have enabled high-school students to very quickly pick up pure functional programming through implementation of animations and games [3].

[1] `http://www.cs.nott.ac.uk/~ggg`

2 Yampa

In the interest of making this paper sufficiently self-contained, we summarize the basics of Yampa in the following. For further details, see earlier papers on Yampa [4,10]. The presentation draws heavily from the Yampa summary in [2].

2.1 Fundamental Concepts

Yampa is based on two central concepts: *signals* and *signal functions*. A signal is a function from time to values of some type:

$$\mathit{Signal}\ \alpha \approx \mathit{Time} \to \alpha$$

Time is continuous, and is represented as a non-negative real number. The type parameter α specifies the type of values carried by the signal. For example, the type of an audio signal, i.e., a representation of sound, would be *Signal Sample* if we take *Sample* to be the type of the varying quantity.[2]

A *signal function* is a function from *Signal* to *Signal*:

$$\mathit{SF}\ \alpha\ \beta \approx \mathit{Signal}\ \alpha \to \mathit{Signal}\ \beta$$

When a value of type $\mathit{SF}\ \alpha\ \beta$ is applied to an input signal of type $\mathit{Signal}\ \alpha$, it produces an output signal of type $\mathit{Signal}\ \beta$. Signal functions are *first class entities* in Yampa. Signals, however, are not: they only exist indirectly through the notion of signal function.

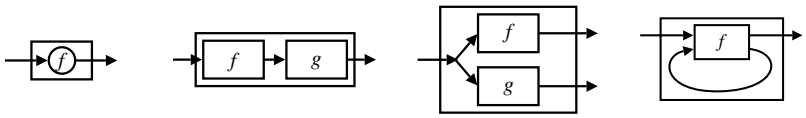
In order to ensure that signal functions are executable, we require them to be *causal*: The output of a signal function at time t is uniquely determined by the input signal on the interval $[0, t]$. If a signal function is such that the output at time t only depends on the input at the very same time instant t, it is called *stateless*. Otherwise it is *stateful*.

2.2 Composing Signal Functions

Programming in Yampa consists of defining signal functions compositionally using Yampa's library of primitive signal functions and a set of combinators. Yampa's signal functions are an instance of the arrow framework proposed by Hughes [7]. Some central arrow combinators are *arr* that lifts an ordinary function to a stateless signal function, composition $\ggg$, parallel composition &&&, and the fixed point combinator *loop*. In Yampa, they have the following types:

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

[2] Physically, sound is varying pressure, and it might come about as a result of the varying displacement of a vibrating string, or the varying voltage of an electronic oscillator. Here we abstract from the physics by referring to the instantaneous value of a sound wave as a "sample", as is conventional in digital audio processing.



(a) *arr f* (b) $f \ggg g$ (c) $f \mathbin{\&\&\&} g$ (d) *loop f*

Fig. 1. Basic signal function combinators

We can think of signals and signal functions using a simple flow chart analogy. Line segments (or "wires") represent signals, with arrowheads indicating the direction of flow. Boxes represent signal functions, with one signal flowing into the box's input port and another signal flowing out of the box's output port. Figure 1 illustrates the aforementioned combinators using this analogy. Through the use of these and related combinators, arbitrary signal function networks can be expressed.

2.3 Arrow Syntax

Paterson's arrow notation [11] simplifies writing Yampa programs as it allows signal function networks to be described directly. In particular, the notation effectively allows signals to be named, despite signals not being first class values. In this syntax, an expression denoting a signal function has the form:

$$
\begin{array}{l}
\textbf{proc}\ pat \rightarrow \textbf{do} \\
\quad pat_1 \leftarrow sfexp_1 \rightarrowtail exp_1 \\
\quad pat_2 \leftarrow sfexp_2 \rightarrowtail exp_2 \\
\quad \ldots \\
\quad pat_n \leftarrow sfexp_n \rightarrowtail exp_n \\
\quad returnA \rightarrowtail exp
\end{array}
$$

Note that this is just *syntactic sugar*: the notation is translated into plain Haskell using the arrow combinators.

The keyword **proc** is analogous to the λ in λ-expressions, *pat* and pat_i are patterns binding signal variables pointwise by matching on instantaneous signal values, *exp* and exp_i are expressions defining instantaneous signal values, and $sfexp_i$ are expressions denoting signal functions. The idea is that the signal being defined pointwise by each exp_i is fed into the corresponding signal function $sfexp_i$, whose output is bound pointwise in pat_i. The overall input to the signal function denoted by the **proc**-expression is bound pointwise by *pat*, and its output signal is defined pointwise by the expression *exp*. An optional keyword **rec** allows recursive definitions (feedback loops).

For a concrete example, consider the following:

$$
\begin{array}{l}
sf = \textbf{proc}\ (a, b) \rightarrow \textbf{do} \\
\qquad (c1, c2) \leftarrow sf1 \mathbin{\&\&\&} sf2 \rightarrowtail a \\
\qquad d \qquad\ \leftarrow sf3 \lll sf4 \rightarrowtail (c1, b)
\end{array}
$$

```
rec
  e ← sf5 -≺ (c2, d, e)
returnA -≺ (d, e)
```

Note the use of the tuple pattern for splitting *sf*'s input into two "named signals", a and b. Also note the use of tuple expressions and patterns for pairing and splitting signals in the body of the definition; for example, for splitting the output from *sf1* &&& *sf2*. Also note how the arrow notation may be freely mixed with the use of basic arrow combinators.

2.4 Events and Event Sources

To model discrete events, we introduce the *Event* type:

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

A signal function whose output signal is of type $\mathit{Event}\ T$ for some type T is called an *event source*. The value carried by an event occurrence may be used to convey information about the occurrence. The operator *tag* is often used to associate such a value with an occurrence:

$$\mathit{tag} :: \mathit{Event}\ a \to b \to \mathit{Event}\ b$$

2.5 Switching

The structure of a Yampa system may evolve over time. These structural changes are known as *mode switches*. This is accomplished through a family of *switching* primitives that use events to trigger changes in the connectivity of a system. The simplest such primitive is *switch*:

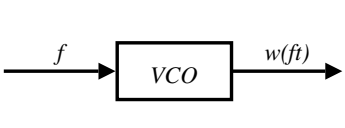
$$\mathit{switch} :: \mathit{SF}\ a\ (b, \mathit{Event}\ c) \to (c \to \mathit{SF}\ a\ b) \to \mathit{SF}\ a\ b$$

The *switch* combinator switches from one subordinate signal function into another when a switching event occurs. Its first argument is the signal function that initially is active. It outputs a pair of signals. The first defines the overall output while the initial signal function is active. The second signal carries the event that will cause the switch to take place. Once the switching event occurs, *switch* applies its second argument to the value tagged to the event and switches into the resulting signal function.

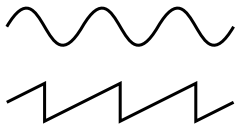
Yampa also includes *parallel* switching constructs that maintain *dynamic collections* of signal functions connected in parallel [10]. We will come back to this when when we discuss how to construct a polyphonic synthesizer.

3 Synthesizer Basics

A modular synthesizer provides a number of sound-generating and sound-shaping modules. By combining these in appropriate ways, various types of sounds can



(a) VCO: f is the control voltage that determines the oscillator frequency; w determines the waveform.



(b) Examples of VCO waveforms.

Fig. 2. Voltage Controlled Oscillator (VCO)

be realized, be they sounds that resemble different acoustic instruments such as string or brass, or completely new ones. Such a configuration of modules is known as a *patch*. Non-modular synthesizers are structured in a similar way, except that the the module configuration to a large extend is predetermined. In this section we introduce some basic synthesizer modules, explain their purpose, and implement some simple ones in Yampa.

3.1 Oscillators

An oscillator is what generates the sound in a synthesizer. As it is necessary to vary the frequency in order to play music, some form of dynamic tuning functionality is needed. Traditionally, this was done by constructing electronic oscillators whose fundamental frequency could be determined by a control voltage. Such a circuit is known as a Voltage Controlled Oscillator (VCO): see Fig. 2(a).

There are many choices for the actual waveform of the oscillator, indicated by the function w in Fig. 2(a). Typically w is some simple periodic function, like the ones in Fig. 2(b): sine and sawtooth. However, w can also be a recording of some sound, often an acoustic instrument. The latter kind of oscillator is the basis of so called sample[3]-based or wavetable synthesizers.

As a first example of using Yampa for sound synthesis, let us implement a simple sine wave oscillator with dynamically controllable frequency. The equations for a sine wave with fixed frequency f are simply

$$\phi = 2\pi f t \tag{1}$$

$$s = sin(\phi) \tag{2}$$

However, we want to allow the frequency to vary over time. To obtain the angle of rotation ϕ at a point in time t, we thus have to *integrate* the varying angular frequency $2\pi f$ from 0 to t. We obtain the following equations:

$$\phi = 2\pi \int_0^t f(\tau)\,\mathrm{d}\tau \tag{3}$$

$$s = sin(\phi) \tag{4}$$

[3] "Sample" is an overloaded term. Depending on context, it can refer either to the sampled, instantaneous value of a signal, or to a recording of some sound. In a digital setting, the latter is a sequence of samples in the first sense.

Let us consider how to realize this. Our sine oscillator becomes a signal function with a control input and an audio output. We will use the type *CV* (for Control Value) for the former, while the type of the latter is just *Sample* as discussed in Sect. 2.1. Further, we want to parameterize an oscillator on its nominal frequency. Thus, our oscillator will become a function that given the desired nominal frequency f_0 returns a signal function whose output oscillates at a frequency f that can be adjusted around f_0 through the control input:

$$\mathit{oscSine} :: \mathit{Frequency} \rightarrow \mathit{SF}\ \mathit{CV}\ \mathit{Sample}$$

Following common synthesizer designs, we adopt the convention that increasing the control value by one unit should double the frequency (up one octave), and decreasing by one unit should halve the frequency (down one octave). If we denote the time-varying control value by $cv(t)$, we get

$$f(t) = f_0 2^{cv(t)} \tag{5}$$

We can now define *oscSine* by transliterating equations 3, 4, and 5 into Yampa code:

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

Note that time is implied, so unlike the equations above, signals are never explicitly indexed by time.

In traditional synthesizers, there is a second class of oscillators known as Low Frequency Oscillators (LFO) which are used to generate time-varying control signals. However, our *oscSine* works just as well at low frequencies. Let us use two sine oscillators where one modulates the other to construct an oscillator with a gentle vibrato:

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

Figure 3 illustrates this patch graphically.

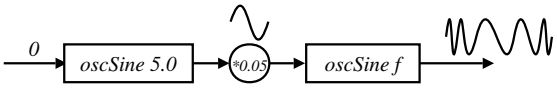


Fig. 3. Modulating an oscillator to obtain vibrato

3.2 Amplifiers

The next fundamental synthesizer module is the variable-gain amplifier. As the gain traditionally was set by a control voltage, such a device is known as a Voltage Controlled Amplifier (VCA). See Fig. 4. VCAs are used to dynamically control the amplitude of audio signals or control signals; that is, multiplication of two signals, where one often is a low-frequency control signal.

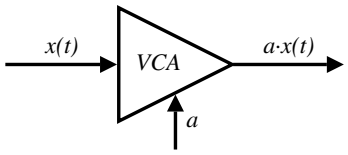


Fig. 4. Voltage Controlled Amplifier (VCA)

An important application of VCAs is to shape the output from oscillators in order to create musical notes with a definite beginning and end. The approach used is to derive a two-level control signal from the controlling keyboard called the *gate* signal. It is typically positive when a key is being pressed and 0 V otherwise. By deriving a second control signal from the keyboard proportional to *which* key is being pressed, feeding this to a VCO, feeding the output from the VCO to the input of a VCA, and finally controlling the gain of the VCA by the gate signal, we have obtained a very basic but usable modular synthesizer patch with an organ-like character: see Fig. 5.

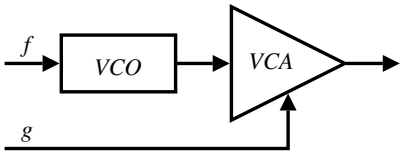
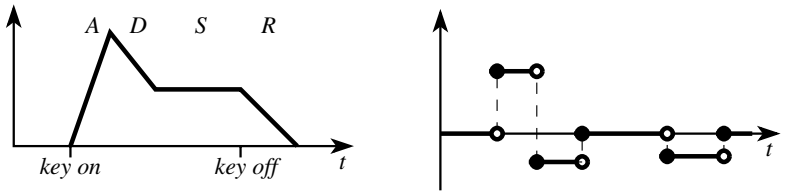


Fig. 5. Basic synthesizer patch: f controls the frequency, g is the gate signal

Since the conceptual operation of a VCA is just multiplication of signals, implementation in Yampa is, of course, entirely straightforward.

3.3 Envelope Generators

When acoustic instruments are played, it often takes a discernable amount of time from starting playing a note until the played note has reached full volume. This is known as the attack. Similarly, a note may linger for a while after the end of the playing action. How the volume of a note evolves over time, its *envelope*, is a very important characteristic of an instrument. In Sect. 3.2, we saw how a patch with an organ-like envelope could be obtained by controlling a VCA with



(a) Classic ADSR envelope (b) Step function

Fig. 6. Envelope generation

the gate signal. To play notes with other types of envelopes, we need to control the VCA with a control signal that mirrors the desired envelope.

An *envelope generator* is a circuit that is designed to allow a variety of musically useful envelopes to be generated. Figure 6(a) shows a classic ADSR envelope. The first phase is the Attack (A). Immediately after a key has been pressed, the control signal grows to its maximal value at a programmable rate. Once the maximal value has been reached, the envelope generator enters the second phase, Decay (D). Here, the control signal decreases until it reaches the sustain level. The third phase is Sustain (S), and the envelope generator will remain there until the key is released. It then enters the fourth phase, Release (R), where the control signal goes to 0. If the key is released before the sustain phase has been reached, the envelope generator will proceed directly to the release phase.

This kind of behaviour is easily programmable in Yampa. An envelope signal with segments of predetermined lengths can be obtained by integrating a step function like the one in Fig. 6(b). Progression to the release phase upon reception of a note-off event is naturally implemented by means of switching from a signal function that describes the initial part of the envelope to one that describes the release part in response to such an event since the release of a key does not happen at a point in time known a priori. Note how the hybrid capabilities of Yampa now start to come in very handy: envelope generation involves both smoothly evolving segments and discrete switching between such segments.

To illustrate, we sketch the implementation of a generalized envelope generator with the following signature:

$$\begin{aligned} &\mathit{envGen} :: \mathit{CV} \to [(\mathit{Time}, \mathit{CV})] \to \mathit{Maybe\ Int} \\ &\qquad \to \mathit{SF}\ (\mathit{Event}\ ())\ (\mathit{CV}, \mathit{Event}\ ()) \end{aligned}$$

The first argument gives the start level of the desired envelope control signal. Then follows a list of time and control-value pairs. Each defines a target control level and how long it should take to get there from the previous level. The third argument specifies the number of the segment before which the sustain phase should be inserted, if any. The input to the resulting signal function is the note-off event that causes the envelope generator to go from the sustain phase to the following release segment(s). The output is a pair of signals: the generated envelope control signal and an event indicating the completion of the

last release segment. This event will often occur significantly *after* the note-off event and is useful for indicating when a sound-generating signal function should be terminated.

Let us first consider a signal function to generate an envelope with a predetermined shape:

```
envGenAux :: CV → [(Time, CV)] → SF a CV
envGenAux l0 tls = afterEach trs >>> hold r0 >>> integral >>> arr (+l0)
  where
    (r0, trs) = toRates l0 tls
```

The auxiliary function *toRates* converts a list of time and level pairs to a list of time and rate pairs. Given such a list of times and rates, the signal function *afterEach* generates a sequence of events at the specified points in time. These are passed through the signal function *hold* that converts a sequence of events, i.e. a discrete-time signal, to a continuous-time signal. The result is a step function like the one shown in Fig. 6(b). By integrating this, and adding the specified start level, we obtain an envelope signal of the specified shape.

We can now implement the signal function *envGen*. In the case that no sustain segment is desired, this is just a matter pairing *envGenAux* with an event source that generates an event when the final segment of the specified envelope has been completed. The time for this event is obtained by summing the durations of the individual segments:

```
envGen l0 tls Nothing = envGenAux l0 tls &&& after (sum (map fst tls)) ()
```

If a sustain segment is desired, the list of time and level pairs is split at the indicated segment, and each part is used to generate a fixed-shape envelope using *envGenAux*. Yampa's *switch* primitive is then employed to arrange the transition from the initial part of the envelope to the release part upon reception of a note-off event:

```
envGen l0 tls (Just n) =
  switch (proc noteoff → do
            l ← envGenAux l0 tls1 -≺ ()
            returnA -≺ ((l, noEvent), noteoff `tag` l))
         (λl → envGenAux l tls2 &&& after (sum (map fst tls2)) ())
  where
    (tls1, tls2) = splitAt n tls
```

Note how the level of the generated envelope signal at the time of the note-off event is sampled and attached to the switch event (the construction *noteoff* `` `tag` `` *l*). This level determines the initial level of the release part of the envelope to avoid any discontinuity in the generated envelope signal.

3.4 A Simple Modular Synthesizer Patch

Let us finish this synthesizer introduction with a slightly larger example that combines most of the modules we have encountered so far. Our goal is a

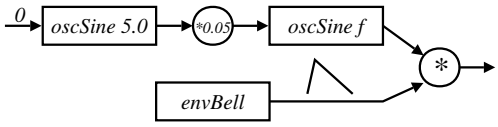


Fig. 7. Vibrato and bell-like envelope

synthesizer patch that plays a note with vibrato and a bell-like envelope (fast attack, gradual decay) in response to events carrying a MIDI note number; i.e., note-on events.

Let us start with the basic patch. It is a function that when applied to a note number will generate a signal function that plays the desired note once:

```
playNote :: NoteNumber → SF a Sample
playNote n = proc _ → do
  v       ← oscSine 5.0          ⤙ 0.0
  s       ← oscSine (toFreq n)   ⤙ 0.05 * v
  (e, _)  ← envBell              ⤙ noEvent
  returnA ⤙ e * s
envBell = envGen 0.0 [(0.1, 1.0), (1.5, 0.0)] Nothing
```

Figure 7 shows a graphical representation of *playNotes*.

The auxiliary function *toFreq* converts from MIDI note numbers to frequency, assuming equal temperament:

```
toFreq :: NoteNumber → Frequency
toFreq n = 440 * (2 ** (((fromIntegral n) − 69.0) / 12.0))
```

Next we need to arrange that to switch into an instance of *playNote* whenever an event carrying a note number is received:

```
playNotes :: SF (Event NoteNumber) Sample
playNotes = switch (constant 0.0 &&& identity)
                   playNotesRec
  where
    playNotesRec n =
      switch (playNote n &&& notYet) playNotesRec
```

The idea here is to start with a signal function that generates a constant 0 audio signal. As soon as a first event is received, we switch into *playNotesRec*. This plays the note once. Meanwhile, we keep watching the input for note-on events (except at time 0, when *notYet* blocks any event as *playNotesRec* otherwise would keep switching on the event that started it), and as soon as an event is received we switch again, recursively, into *playNotesRec*, thus initiating the playing of the next note. And so on.

4 A SoundFont-Based Monophonic Synthesizer

The SoundFont format is a standardized way to describe musical instruments. It is a sample-based format, i.e. based on short recordings of actual instruments, but it also provides true synthesizer capabilities through a network of interconnected modules of the kind described in Sect. 3. In this section, we sketch how to turn a SoundFont description into a monophonic synthesizer using Yampa.

4.1 Implementing a Sample-Based Oscillator

We first turn our attention to implementing an oscillator that uses recorded waveforms or samples. A SoundFont file contains many individual samples (often megabytes of data), each a recording of an instrument playing some particular note. Along with the actual sample data there is information about each sample, including the sampling frequency, the fundamental (or *native*) frequency of the recorded note, and loop points. The latter defines a region of the sample that will be repeated to allow notes to be sustained. Thus samples of short duration can be used to play long notes.

In our implementation, data for all the samples is stored in a single array:

```
type SampleData = UArray SamplePointIndex Sample
type SamplePointIndex = Word32
```

Note that the type *Sample* here refers to an instantaneous sample value, as opposed to an entire recording. Information about individual samples are stored in records of type *SampleInfo*. In addition to the information already mentioned, these also store the start and end index for each sample.

A sample-playing oscillator can now be defined in much the same way as the sine oscillator from Sect. 3.1, the main difference being that the periodic function now is given by table lookup and linear interpolation:

```
oscSmplAux :: Frequency → SampleData → SampleInfo
  → SF CV (Sample, SamplePointIndex)
oscSmplAux freq sdta sinf = proc cv → do
  phi ← integral⤙ freq / (smplFreq sinf) * (2 ** cv)
  let (n, f) = properFraction (phi * smplRate sinf)
    p1 = pos n
    p2 = pos (n + 1)
    s1 = sdta ! p1
    s2 = sdta ! p2
  returnA⤙ (s1 + f * (s2 - s1), p2)
  where
    pos n = ...
```

The local function *pos* converts a sample number to an index by "wrapping around" in the loop region as necessary. In addition to the instantaneous sample value, the oscillator also outputs the current sample index. This enables a smooth

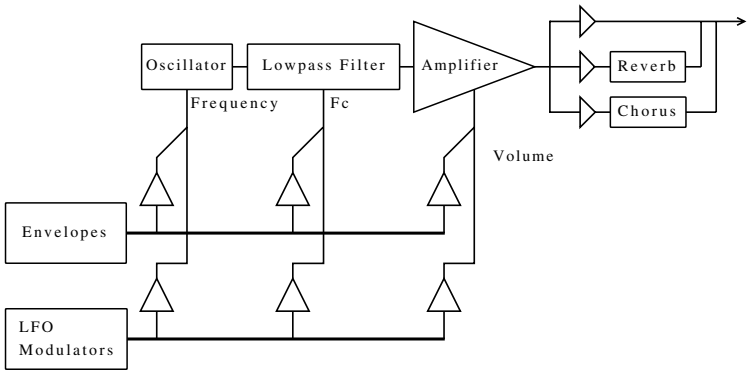


Fig. 8. The SoundFont synthesis model

transition to the release segment of a sample (after the loop region) when a note-off event is received.

Finally, we can define a complete oscillator that takes care of the transition to the release segment on a note-off event through switching from *oscSmplAux* to an oscillator that plays the release segment. We only give the type signature:

$$oscSmpl :: Frequency \rightarrow SampleData \rightarrow SampleInfo \rightarrow SF\ (CV, Event\ ())\ Sample.$$

4.2 Combining the Pieces

Given the sample-based oscillator, a complete SoundFont synthesizer can be obtained by wiring together the appropriate modules according to the SoundFont synthesis model shown in Fig. 8, just like the simple monophonic synthesizer was constructed in Sect. 3.4. The SoundFont model does include *filters*. While not considered in this paper, filters can easily be realized using Yampa's unit delays [12].

In our case, we also choose to do the MIDI processing at this level. Each monophonic synthesizer is instantiated to play a particular note at a particular MIDI channel at some particular strength (velocity). The synthesizer instance continuously monitors further MIDI events in order to identify those relevant to it, including note-off events to switch to the release phase and any articulation messages like pitch bend. This leads to the following type signature, where the output event indicates that the release phase has been completed and the playing of a note thus is complete:

$$\textbf{type}\ MonoSynth = Channel \rightarrow NoteNumber \rightarrow Velocity \rightarrow SF\ MidiEvent\ (Sample, Event\ ()).$$

5 A Polyphonic Synthesizer

In this section, we consider how to leverage what we have seen so far in order to construct a polyphonic synthesizer capable of playing standard MIDI files.

5.1 Dynamic Synthesizer Instantiation

The central idea is to instantiate a monophonic synthesizer in response to every note-on event, and then run it in parallel with any other active synthesizer instances until the end of the played note. Yampa's parallel switching construct [10] is what enables this dynamic instantiation:

```
pSwitchB :: Functor col ⇒
  col (SF a b)                                -- Initial signal func. collection
  → SF (a, col b) (Event c)                   -- Event source for switching
  → (col (SF a b) → c → SF a (col b))         -- Signal function to switch into
  → SF a (col b)
```

The combinator *pSwitchB* is similar to *switch* described in Sect. 2.5, except that

- a collection of signal functions are run in parallel
- a separate signal function is used to generate the switching event
- the function computing the signal function to switch into receives the collection of subordinate signal functions as an extra argument.

The latter allows signal functions to be *independently* added to or removed from a collection in response to note-on and monosynth termination events, while *preserving* the state of all other signal functions in the collection.

The overall structure of the polyphonic synthesizer is shown in Fig. 9. The signal function *triggerChange* generates a switching event when reconfiguration is necessary (i.e. when adding or removing monosynth instances). The function *performChange* computes the new collection of monosynth instances after a switch. The output signal from the parallel switch is a collection of samples at each point in time, one for every running monosynth instance. This can be

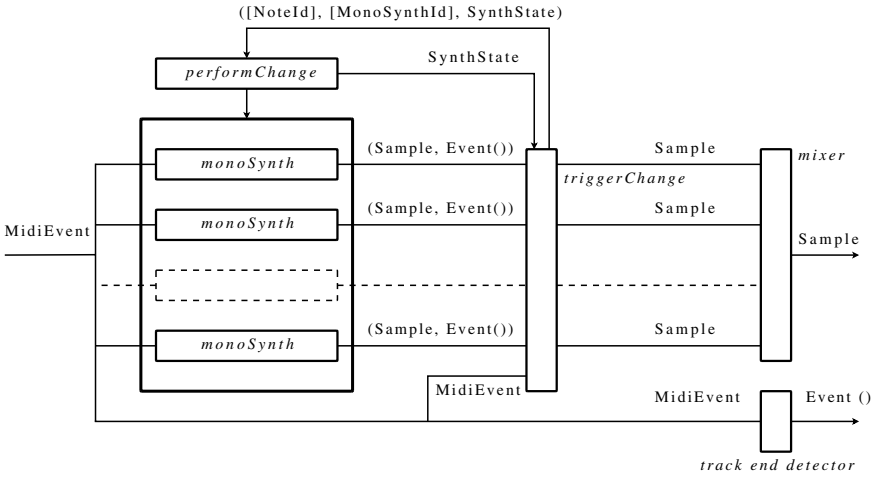


Fig. 9. Overall structure of the polyphonic synthesizer

seen as a collection of audio signals. The signal function *mixer* sums all these samples into a single audio signal.

5.2 Performance

Despite being implemented in a very straightforward (almost naive) way, the performance of the polyphonic synthesizer is reasonable. For example, using modern hardware (1.8 GHz Intel dual core) and compiling using GHC, a moderately complex score like Mozart's *Rondo Alla Turca*, can be rendered as fast as it plays at 22 kHz sampling rate using a SoundFont[4] piano definition. However, an audio buffer is needed between the synthesizer process and the audio player to guard against garbage collection pauses and the like: thus, the *latency* is high.

6 Related Work

Haskore [5] is a language for programming music embedded in Haskell. Its fundamental design resembles traditional musical scores, but as it is an embedding in Haskell, Haskell can be used for "meta programming". Haskore itself does not deal with defining instruments, but see the discussion of HasSound below. Describing musical scores was not our focus in this work. Haskore could clearly be used to that end, being a Haskell embedding just like Yampa. Since our framework provides an interface to MIDI and MIDI files, *any* application capable of generating MIDI could in principle be used as a frontend. However, one could also explore implementing "score-construction" abstraction directly in the Yampa framework. An interesting aspect of that would be that there is no firm boundary between the musical score and the sounds used to perform it. One could also imagine interactive compositions, as Yampa is a reactive programming language.

Csound is a domain-specific language for programming sound and musical scores [14]. Fundamentally, it is a modular synthesizer, enabling the user to connect predefined modules in any conceivable manner. It is possible to extend Csound with new modules, but these have to be programmed in the underlying implementation language: C. Thanks to its extensibility, Csound now provides a vast array of sound generating and sound shaping modules. Obviously, what we have done in this paper does not come close to this level of maturity. However, we do claim that our hybrid setting provides a lot of flexibility in that it both allows the user to implement basic signal generation and processing algorithms as well as higher-level discrete aspects in a single framework, with no hard boundaries between the levels.

HasSound [6] is a domain-specific language embedded in Haskell for defining instruments. It is actually a high-level frontend to Csound: HasSound definitions are compiled to Csound instrument specifications. Therein lies both HasSound's strength and weakness. On the one hand, HasSound readily provides access to lots of very sophisticated facilities from Csound. On the other hand, the end

[4] http://www.sf2midi.com/index.php?page=sdet&id=8565

result is ultimately a static Csound instrument definition: one cannot do anything in HasSound that cannot (at least in principle) be done directly in Csound. The approach taken in this paper is, in principle, more flexible.

Low-level audio processing and sound-generation in Haskell has also been done earlier. For example, Thielemann [13] develops an audio-processing framework based on representing signals as co-recursively defined streams. However, the focus is on basic signal processing, not on synthesis applications.

Karczmarczuk [8] presents a framework with goals similar to ours using a stream-based signal representation. Karczmarczuk focuses on musically relevant algorithms and present a number of concise realizations of physical instrument simulations, including the Karplus-Strong model of a plucked string [9], reverb, and filters. He also presents an efficient, delay-based sine oscillator, and does consider how to modulate its frequency by another signal to create vibrato.

However, Karczmarczuk's framework, as far as it was developed in the paper, lacks the higher-level, discrete facilities of Yampa, and the paper does not consider how to actually go about programming the logic of playing notes, adding polyphony[5], etc. Also, the arrow framework offers a very direct and intuitive way to combine synthesizer modules: we dare say that someone familiar with programming modular synthesizers would feel rather at home in the Yampa setting, at least as long as predefined modules are provided. The correspondence is less direct in Karczmarczuk's framework as it stands.

7 Conclusions

FRP and Yampa address application domains that have not been traditionally associated with pure declarative programming. For example, in earlier work we have applied Yampa to video game implementation [2], and others have since taken those ideas much further [1]. In this paper, we have applied Yampa to another domain where pure declarative programming normally is not considered, modular synthesis, arguing that the hybrid aspects of Yampa provides a particularly good fit in that we can handle both low-level signal processing and higher-level discrete aspects, including running many synthesizer instances in parallel to handle polyphony. We saw that Yampa's parallel, collection-based switch construct [10] was instrumental for achieving the latter. We also think that being able to do all of this seamlessly in a single framework opens up interesting creative possibilities.

As it stands, our framework is mainly a proof of concept. Nevertheless, we feel that the Yampa style of programming is immediately useful in an educational context as it makes it possible to implement interesting examples from somewhat unexpected domains in an intuitive, concise, and elegant manner, thus providing an incentive to learn pure declarative programming. We note that others have had similar experiences with related approaches [3].

[5] Summing a fixed number of streams to play more than one note is, of course, straightforward. But polyphonic performance requires independent starting and stopping of sound sources.

References

1. Cheong, M.H.: Functional programming and 3D games. In: BEng thesis, University of New South Wales, Sydney, Australia (November 2005)
2. Courtney, A., Nilsson, H., Peterson, J.: The Yampa arcade. In: Haskell 2003. Proceedings of the 2003 ACM SIGPLAN Haskell Workshop, Uppsala, Sweden, pp. 7–18. ACM Press, New York (2003)
3. Felleisen, M.: Personal communication and on-line lecture notes (June 2007), http://www.ccs.neu.edu/home/matthias/HtDP/Prologue/book.html
4. Hudak, P., Courtney, A., Nilsson, H., Peterson, J.: Arrows, robots, and functional reactive programming. In: Jeuring, J., Peyton Jones, S.L. (eds.) AFP 2002. LNCS, vol. 2638, pp. 159–187. Springer, Heidelberg (2003)
5. Hudak, P., Makucevich, T., Gadde, S., Whong, B.: Haskore music notation - an algebra of music. Journal of Functional Programming 6(3), 465–483 (1996)
6. Hudak, P., Zamec, M., Eisenstat, S.: HasSound: Generating musical instrument sounds in Haskell. NEPLS talk, Brown University. Slides (October 2005), http://plucky.cs.yale.edu/cs431/HasSoundNEPLS-10-05.ppt
7. Hughes, J.: Generalising monads to arrows. Science of Computer Programming 37, 67–111 (2000)
8. Karczmarczuk, J.: Functional framework for sound synthesis. In: Hermenegildo, M.V., Cabeza, D. (eds.) PADL 2005. LNCS, vol. 3350, pp. 7–21. Springer, Heidelberg (2005)
9. Karplus, K., Strong, A.: Digital synthesis of plucked string and drum timbres. Computer Music Journal 7(2), 43–55 (1983)
10. Nilsson, H., Courtney, A., Peterson, J.: Functional reactive programming, continued. In: Haskell 2002. Proceedings of the 2002 ACM SIGPLAN Haskell Workshop, Pittsburgh, Pennsylvania, USA, pp. 51–64. ACM Press, New York (2002)
11. Paterson, R.: A new notation for arrows. In: Proceedings of the 2001 ACM SIGPLAN International Conference on Functional Programming, Firenze, Italy, pp. 229–240 (September 2001)
12. Smith, J.O.: Introduction to Digital Filters, August 2006 edn. CCRMA (May 2006), http://ccrma.stanford.edu/~jos/filters06/
13. Thielemann, H.: Audio processing using Haskell. In: DAFx 2004. Proceedings of the 7th International Conference on Digital Audio Effects, Naples, pp. 201–206 (2004)
14. Vercoe, B.: The Canonical Csound Reference Manual. MIT Media Lab (2007)