Thesis for the Degree of Master of Science

Modular Synthesizer Programming in Haskell

George Giorgidze

CHALMERS | GÖTEBORG UNIVERSITY

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg Sweden

Göteborg, February 2008

Modular Synthesizer Programming in Haskell
George Giorgidze

Examiner: Koen Claessen

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Printed at Chalmers, Göteborg, Sweden, 2008

Abstract

In this thesis, 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 an audio with respectable performance.

Declaration

I implemented the software synthesizer and its supporting libraries described in the thesis. The source code is available online[1] under BSD3 license. This report is based on [Giorgidze and Nilsson, 2008] written in collaboration by my co-author and me. For this thesis it was modified and extended by me. Sections and figures adapted from other sources are clearly indicated in respective chapters and original sources are referenced.

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

Acknowledgements

I would like to gratefully acknowledge Henrik Nilsson, for giving me an opportunity to work on the project I was interested in, and for his contribution to original article [Giorgidze and Nilsson, 2008] of which modified and extended version is this report.

I would also like to thank lecturers and tutors from Chalmers University of Technology, in no particular order Björn Bringert, Koen Claessen, Aarne Ranta and David Sands. These are the people that have drastically changed my views about programming languages, by teaching Haskell and functional programming.

I thank my colleague and office mate Neil Sculthorpe for fruitful discussions on Yampa related topics, and for a detailed feedback on the thesis draft. Thank you Neil!

My friends and lab partners from our Master's program: Magnus Karlsson, Thomas Schilling, Kai Wang, Kirubel Tekle, Andrey Chudnov, Arun Reddy and Moises Salvador Meza Morena deserve a special acknowledgement for making my studies at Chalmers and my stay in Göteborg a truly memorable experience.

Lastly, I acknowledge the most important person in my life, my wife Mari Chikvaidze, for her constant love, support and encouragement. This thesis is dedicated to her.

Dedicated to Mari

Contents

Chapter 1

Introduction

In this thesis project we have implemented a modular synthesizer in the purely functional programming language, Haskell. The project is built on the top of Yampa, a domain specific language embedded in Haskell. In the context of this report, a synthesizer is a reactive system, hardware device or software program, which generates an audio signal in response to received input. We develop a software synthesizer, though we often compare our implementation to hardware variants or even use hardware implementations as a guideline.

Yampa provides a declarative framework for programming hybrid systems. As music can be seen as a hybrid phenomenon, it is interesting to apply Yampa to musical applications.

In summary, the Yampa approach to programming hybrid systems is demonstrated and evaluated by developing a synthesizer gradually, starting from basic synthesizer components and ending with a MIDI music synthesizer capable of rendering audio in real-time with acceptable quality.

1.1 Music as a Hybrid Phenomenon

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: it consists of sequences of discrete notes. Figure 1.1 illustrates this point. In addition a musical performance can have some continuous elements as well. For example the aforementioned score might contain instructions such as: gradually increase tempo, and decrease loudness.

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.

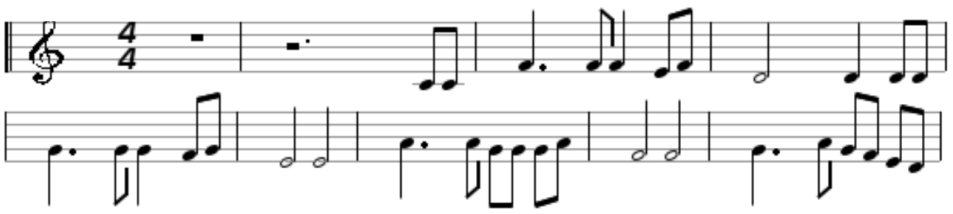


Figure 1.1: Traditional Musical Score

Traditional musical notation, or its modern-day electronic derivatives such as Musical Instrument Digital Interface (MIDI) files or domain-specific languages like Haskore [Hudak et al., 1996], 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 [Vercoe, 2007]. 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. Moreover, the facilities offered at the sound-programming level normally fall short of a general-purpose programming language, except that it may be possible to write extensions in an underlying low-level (from the application point of view) implementation language like C.

Here we aim to describe both continuous and discrete aspects of music and musical applications in a single framework. Yampa [Hudak et al., 2003, Nilsson et al., 2002], an instance of the Functional Reactive Programming (FRP) paradigm in the form of a domain-specific language embedded in Haskell, supports programming of hybrid systems, and provides the necessary language features to accomplish the goal.

1.2 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, in a same way as early analogue computers were "programmed".

Figure 1.2 illustrates sound-generating and sound-shaping modules wired together with so called *patch-cords*. The whole configuration is referred to as a *patch* of modular synthesizers. Here we illustrate the software which is used for the configuration. If synthesizer is constructed entirely from

hardware without any software components, then electrical wires are used to configure it.

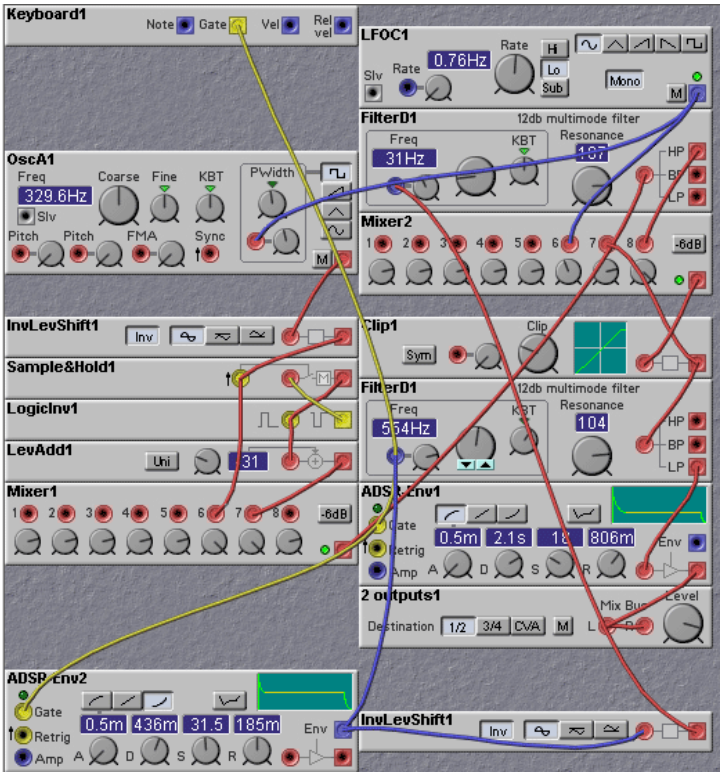


Figure 1.2: Screenshot from Clavia Nord Modular G2's patch editor

Usually a keyboard (see Figure 1.3) is used to control modular synthesizer musically and to realize sounds for which it was programmed. Most basic keyboards are capable of generating discrete *NoteOn* and *NoteOff* events, however it might also have some continuous controllers as well, generating continuous signals. These signals might represent the position of pitch wheel, pressure on the key after depression, etc.

Figure 1.3: Musical Keyboard

Nowadays, because of its flexibility, modular synthesis is used both in hardware and software synthesizers. We also follow a similar approach when constructing a synthesizer in Yampa.

1.3 Contributions

In this report, 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 descriptions 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 as fast as it plays.

All code is available on-line[1] under BSD3 license.

In addition, the code includes supporting library for reading, writing and manipulating of MIDI, SoundFont and Waveform audio (WAV) multimedia file formats, entirely implemented in Haskell. These libraries serve as a supporting infrastructure to our framework, but it is general enough to be used in a different context as well.

The contribution of this work lies in the application of declarative hybrid programming to a novel application area, and serves 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.

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

Chapter 2

Yampa

In the interest of making this report self-contained, we summarize the basics of Yampa in the following. For further details, see earlier papers and reports on Yampa [Hudak et al., 2003, Nilsson et al., 2002, Sculthorpe, 2007]. The presentation draws heavily from the Yampa summary in [Courtney et al., 2003].

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 α:

$$Signal\ \alpha \approx Time \rightarrow \alpha$$

Time is continuous, and is represented as a non-negative real number[1]. 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*:

$$SF\ \alpha\ \beta \approx Signal\ \alpha \rightarrow Signal\ \beta$$

When a value of type $SF\ \alpha\ \beta$ is applied to an input signal of type $Signal\ \alpha$, it produces an output signal of type $Signal\ \beta$. 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

[1] In current implementation of Yampa, *Time* is synonym of Haskell's *Double* type

[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. In the current implementation, *Sample* is of *Double* type in Haskell.

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*.

For example let us consider signal function $identity :: SF\ a\ a$, its output signal is always the same as input signal at any time t and obviously only depends on input at t, hence *identity* is a stateless signal function. In contrast $integral :: Floating\ a \Rightarrow SF\ a\ a$ is a stateful signal function, because it outputs integral of input signal, which at time t depends on input in the interval of $[0, t]$.

Signals and signal functions are often represented as diagrams. Figure 2.1 illustrates one such diagram.

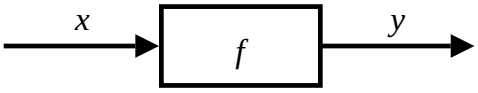


Figure 2.1: A Signal Function, $SF\ a\ b$

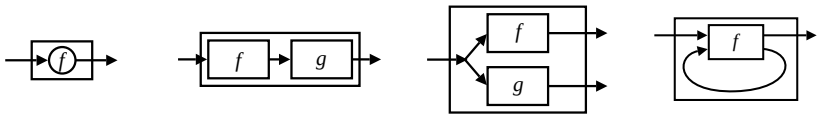
Boxes in a Figure 2.1 represent signal functions with one signal flowing into the box's input port and another signal flowing out of the box's output port. Line segments (or "wires") represent signals. Arrow heads are used to indicate direction of flow. So $f :: SF\ a\ b$ is a signal function, and $x :: Signal\ a$ and $y :: Signal\ b$ are signals.

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 [Hughes, 2000]. Some central arrow combinators are *arr* that lifts an ordinary function to a stateless signal function, $\ggg$, &&&, and *loop*. In Yampa, they have the following type signatures:

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

Figure 2.2 illustrates aforementioned combinators using diagrams. Through the use of these and related combinators, arbitrary signal function networks can be expressed.



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

Figure 2.2: Basic signal function combinators.

2.3 Arrow Syntax

Paterson's arrow notation [Paterson, 2001] 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:

proc *pat* $\rightarrow$ **do**
  $pat_1 \leftarrow sfexp_1 \prec exp_1$
  $pat_2 \leftarrow sfexp_2 \prec exp_2$
  ...
  $pat_n \leftarrow sfexp_n \prec exp_n$
  $returnA \prec exp$

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*. The signal variables bound in the patterns may occur in the signal value expressions, but *not* in the signal function expressions $sfexp_i$.

An optional keyword **rec**, applied to a group of definitions, permits signal variables to occur in expressions that textually precede the definition of the variable, allowing recursive definitions (feedback loops). Finally,

let *pat* = *exp*

is shorthand for

$pat \leftarrow arr\ id \prec exp$

allowing binding of instantaneous values in a straightforward way.

The syntactic sugar is implemented by a preprocessor which expands out the definitions using only the basic arrow combinators *arr*, $\ggg$, *first*, and, if **rec** is used, *loop*.

Here we use $first :: SF\ a\ b \to SF\ (a, c)\ (b, c)$ as primitive combinator (see Figure 2.3) instead of &&&, they can be defined in terms of each other, so one or another can be used as primitive without loosing expressiveness.

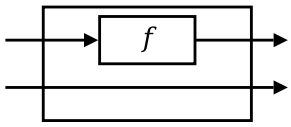


Figure 2.3: Signal Function combinator *first*

For a concrete example, consider the following:

$$
\begin{array}{l}
sf = \mathbf{proc}\ (a, b) \to \mathbf{do} \\
\quad (c1, c2) \leftarrow sf1 \mathbin{\&\&\&} sf2 \rightarrowtail a \\
\quad d \quad\quad \leftarrow sf3 \lll sf4 \rightarrowtail (c1, b) \\
\quad \mathbf{rec} \\
\quad\quad e \leftarrow sf5 \rightarrowtail (c2, d, e) \\
\quad returnA \rightarrowtail (d, e)
\end{array}
$$

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.

For illustration purposes we translate the code above to plain arrow combinators.

$$
\begin{array}{l}
sf \\
\quad = ((first\ (sf1 \mathbin{\&\&\&} sf2) \ggg arr\ (\lambda((c1, c2), b) \to ((b, c1), c2))) \\
\quad\quad \ggg \\
\quad\quad (first\ (arr\ (\lambda(b, c1) \to (c1, b)) \ggg (sf3 \lll sf4)) \ggg \\
\quad\quad\quad loop \\
\quad\quad\quad\quad (arr\ (\lambda((d, c2), e) \to ((c2, d, e), d)) \ggg \\
\quad\quad\quad\quad\quad (first\ sf5 \ggg arr\ (\lambda(e, d) \to ((d, e), e))))))
\end{array}
$$

Even for moderately complexed networks, the combinator notation becomes very hard to read. In contrast the arrow syntactic sugar provides a clearer and more intuitive way to describe signal function networks.

2.4 Events and Event Sources

While some aspects of a program (such as sound) are naturally modelled as continuous signals, other aspects (such as a key on an organ being pressed and subsequently released) are more naturally modelled as *discrete events*.

To model discrete events, Yampa introduces the *Event* type, which is isomorphic to Haskell's *Maybe* 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 function *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.

Informally, *switch sf sfk* behaves as follows: At time $t = 0$, the initial signal function, *sf*, is applied to the input signal of the *switch* to obtain a pair of signals, *bs* (type: *Signal bs*) and *es* (type: *Signal* (*Event c*)). The output signal of the *switch* is *bs* until the event stream *es* has an occurrence at some time t_e, at which point the event value is passed to *sfk* to obtain a signal function sf'. The overall output signal switches from *bs* to sf' applied to a suffix of the input signal starting at t_e.

Yampa also includes *parallel* switching constructs that maintain *dynamic collections* of signal functions connected in parallel [Nilsson et al., 2002]. Signal functions can be added to or removed from such a collection at run-time in response to events, while *preserving* any internal state of all other signal functions in the collection. See Figure 2.4.

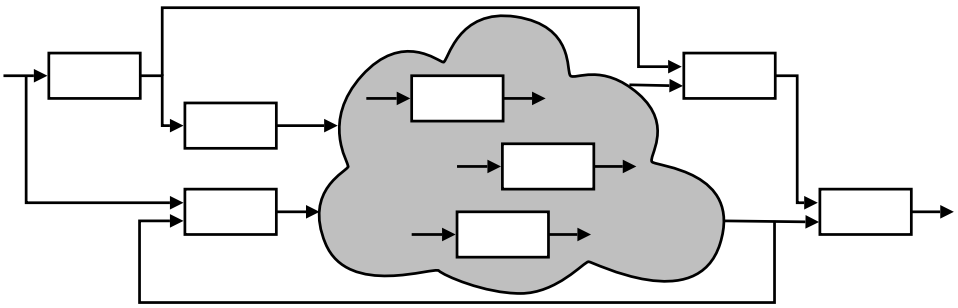


Figure 2.4: System of interconnected signal functions with varying structure

The first class status of signal functions in combination with switching over dynamic collections of signal functions makes Yampa an unusually flexible language for describing hybrid systems, with highly dynamic structure. In the following we will see that this capability is central for constructing a polyphonic synthesizer in a natural and elegant way.

2.6 Animating Signal Functions

Thus far we have seen simple declarative specifications of interactive objects as signal functions. One notable omission from these specifications was any explicit mention of the I/O system or a connection to the external world. This is quite deliberate, and is a hallmark of Yampa programming. Instead of specifying an interactive application as an explicit sequence of I/O actions, the Yampa programmer defines a signal function that transforms an input signal of some type into an output signal of some type , compositionally using the Yampa primitives and arrow combinators. This programming style ensures that Yampa programs have a simple, precise denotation independent of the (typically complex and underspecified) details of the I/O system or the external world.

To actually execute a Yampa program, i.e. the top-level signal function representing an entire system, we need some way to connect the program's input and output signals to the external world. Yampa provides the function *reactimate* for this purpose [3].

$$
\begin{array}{ll}
\mathit{reactimate} :: \mathit{IO}\ (\mathit{DTime}, a) & \text{-- Sense} \\
\quad \rightarrow (b \rightarrow \mathit{IO}\ ()) & \text{-- Actuate} \\
\quad \rightarrow \mathit{SF}\ a\ b & \\
\quad \rightarrow \mathit{IO}\ () &
\end{array}
$$

[3] for presentation purposes here we present slightly simplified version of *reactimate* function. Please see the source code of the application for further details.

Reactimate approximates the continuous-time model presented here by performing discrete sampling of the signal function, feeding input to and processing output from the signal function at each time step. The first argument to *reactimate* is an IO action that will obtain the next input value along with the amount of time elapsed (or "delta" time, *DTime*) since the previous sample. For example, in our setting, for CD audio quality, the delta time would correspond to a system sampling frequency of 44.1 kHz, i.e. 1/44100 s, and the input could be note-on and note-off messages from a connected synthesizer keyboard. The second argument is a function that, given an output value, produces an IO action that will process the output in some way. For example, it could send sound samples to the audio subsystem for immediate playback, or record them to an audio file for later playback. The third argument is the signal function to be animated.

Chapter 3

Synthesizer Basics

A modular synthesizer provides a number of sound-generating and sound-shaping modules. By combining these modules in appropriate ways, various types of sounds can 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 extent is predetermined. They effectively come pre-patched from the factory. While this obviously implies a certain lack of flexibility, it does make the synthesizer easier to use, and in the early days of synthesizers, a more or less fixed configuration was also what made it possible to build affordable, portable, and road-worthy instruments.

As important as generating a particular sound, is to give the performer dynamic control over that sound in musically meaningful ways. The most obvious is that it must be possible to vary the frequency in order to play specific notes, and to have control over when a note starts and ends. Other forms of articulation are also important, such as varying the loudness or vibrato.

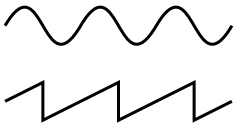
In this section we introduce some basic synthesizer modules, explain their purpose, and implement some simple ones in Yampa, and explain how they can be connected into a very rudimentary monophonic synthesiser.

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 Figure 3.1(a). A typical convention is that an increase of the control voltage by 1 V causes the frequency of the oscillator to double; i.e.,

up one octave. Even today, in pure digital or software synthesizers, this term VCO is still in widespread use, even if the physical realization is very far from the original analogue circuity.

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



(b) Examples of VCO waveforms.

Figure 3.1: Voltage Controlled Oscillator (VCO)

There are many choices for the actual waveform of the oscillator, indicated by the function w in Figure 3.1(a). Typically w is some simple periodic function, like the ones in Figure 3.1(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[1] - 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

$$\phi = 2\pi f t \tag{3.1}$$
$$s = sin(\phi) \tag{3.2}$$

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:

$$\phi = 2\pi \int_0^t f(\tau)\,\mathrm{d}\tau \tag{3.3}$$
$$s = sin(\phi) \tag{3.4}$$

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 Section 2.1. Further, we want to parametrize an oscillator on its nominal frequency. Thus, our oscillator will become a function that given

[1] "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.

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: Here is the type signature:

$$oscSine :: Frequency \rightarrow SF\ CV\ Sample$$

The types *Sample* and *CV* are only advisory type synonyms for *Double*.

What should the relation between the nominal frequency f_0 and the varying frequency f be? 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

$$f(t) = f_0 2^{cv(t)} \tag{3.5}$$

We can now define *oscSine* by transliterating equations 3.3, 3.4, and 3.5 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. Thus *cv*, *f*, and *phi* all refer to whatever value the corresponding signal has at some particular point in time. This is similar to how differential equations are usually stated. The integration limits are also implied: the output from *integral* is the integral of the input from 0 to the present point in time.

While simple, *oscSine* is a perfectly usable audio oscillator. 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. This was done in part because the low frequencies necessitated different circuit solutions. 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:

```
constant 0 >>> oscSine 5.0 >>> arr (*0.05) >>> oscSine 440
```

Figure 3.2 illustrates this patch graphically.

In modular synthesis, it is common to modulate both with low-frequency control signals and with audio signals. Indeed, nothing stops us from running the first oscillator at audio frequencies too. This tends to create waveforms very rich in overtones and is the basis of Frequency Modulation (FM) synthesis, as implemented in the famous Yamaha DX7 synthesizer and its siblings.

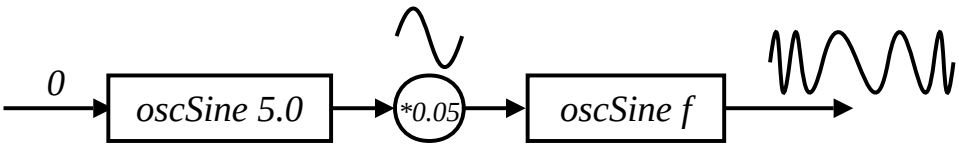


Figure 3.2: Modulating an oscillator to obtain vibrato

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 Figure 3.3. This term is frequently used also for digital or software implementations. 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.

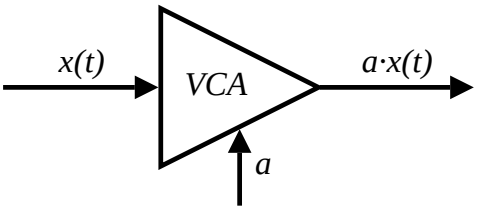


Figure 3.3: Voltage Controlled Amplifier (VCA)

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 Figure 3.4.

By feeding the output through a second VCA, and controlling that VCA with a suitably scaled and offset signal from an LFO, we can add a gentle tremolo (amplitude modulation) to our sound.

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

$$vca :: SF\ (Sample, CV)\ Sample$$
$$vca = arr\ (curry\ (*))$$

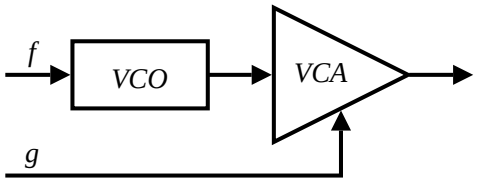


Figure 3.4: Basic synthesizer patch: f controls the frequency, g is the gate signal.

In fact, there is not even much point in introducing a name for such a simple definition, especially not when programming using the arrow notation. However, it is common to want to set the overall loudness of a note according to how forcefully a key was being played. In practice, what is measured is how quickly a key was being pressed down, usually referred to as the *key-on velocity*. Thus we introduce an amplifier module that sets the overall gain according to the key-on velocity, and then allows that gain to be dynamically changed by a control signal:

$$
\begin{aligned}
&amp :: Velocity \to SF\ (Sample, CV)\ Sample \\
&amp\ vel = \mathbf{proc}\ (sample, cv) \to \mathbf{do} \\
&\quad returnA \prec (vel' \,/\, 127.0) * (sample * (cv + 1.0)) \\
&\quad \mathbf{where}\ vel' = fromIntegral\ vel
\end{aligned}
$$

We follow the MIDI standard and represent Velocity by an integer between 0 and 127.

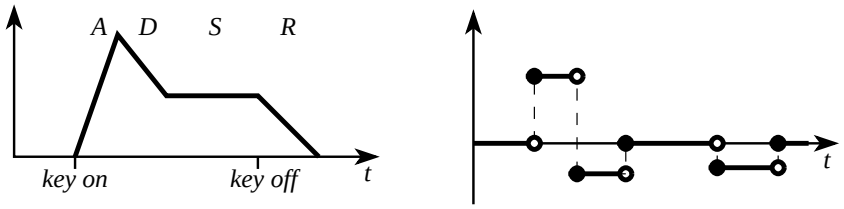
There is an implied assumption here (by referring to the overall gain as "velocity") that we will instantiate a new amplifier every time a key is pressed. This is of course very unlike what would be going on in a hardware synthesizer! But in a software implementation, it is both feasible and useful to instantiate oscillators, amplifiers, etc. in response to key-on and key-off messages, and that is what we eventually are going to do.

3.3 Envelope Generators

When acoustic instruments are played, it often takes a discernible 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. Also, some instruments are capable of playing sustained notes (like flutes, organs), whereas the sound from others will fade away, or decay, more or less quickly after the initial playing action (like pianos, bells). In all, how the volume of a note evolves over time, its *envelope*, is a very important characteristic of an instrument. In Section 3.2, we saw how a patch with an organ-like envelope (i.e., from no sound to full volume as soon as a key is pressed, and from full volume to no sound equally

quickly when the key is released) could be obtained by controlling a VCA with 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. The kind of control signals obtainable form a classic ADSR envelope generator is illustrated in Figure 3.5(a).



(a) Classic ADSR envelope (b) Step function

Figure 3.5: Envelope generation

There are four phases. 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.

Besides controlling how the volume of a note evolves, envelope generators have plenty of other applications where gradual changes of some parameter are desired. 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 Figure 3.5(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.

We illustrate the implementation of a generalized envelope generator with the following signature:

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

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 Figure 3.5(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 Filters

Another important module for modular synthesis is the filter. The idea is to start with a waveform with a rich overtone spectra, like triangle, sawtooth, and pulse, and then apply a filter to attenuate certain overtones, accentuate others, and maybe even add new ones if the filter has a sufficiently resonant character. This is known as *subtractive* synthesis. The most commonly employed filter is the Low-pass Filter (LP), which attenuates frequencies above a certain characteristic frequency known as the corner frequency, f_c. Many other filter types are also employed, but we will only consider the LP filters in the following.

The frequency f_c must be dynamically controllable. Otherwise a low frequency tone from the oscillator would get a very different character from a high-frequency tone once it has passed the filter. Indeed, if the latter frequency is much larger than f_c, we may hear nothing at all! Furthermore, a classic and much loved synthesizer effect involves sweeping the corner frequency f_c while a note is sounding, thus dynamically altering the character of the tone. Dynamic control of f_c is achieved through a control voltage: hence synthesizer filters are often referred to as Voltage Controlled Filters, or VCFs.

The filter is arguably one of the most important factors in determining the sonic character of a synthesizer, and some filter designs, like the Moog transistor ladder filter, have achieved legendary status

Digital realizations of filter usually involves one or more unit delays [Smith, 2006a]. Figure 3.6 shows the simplest possible digital low-pass filter, where z^{-1} denotes the unit delay.

This low-pass filter is readily implementable in Yampa:

```
(identity &&& iPre 0) >>> arr (uncurry (+))
```

The combinator *iPre* is Yampa's (initialized) unit delay.

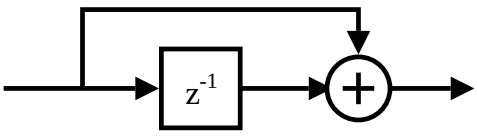


Figure 3.6: Simplest possible digital low-pass filter

Note that we now, for the first time, are *exploiting* the fact that Yampa fundamentally is a discretely sampled (see Section 2.6). In most applications that is something we would try to avoid: it is useful to think of time as being truly continuous, and not predicating the behaviour of a system on any particular sampling interval as long as the sampling is "sufficiently frequent" [Wan and Hudak, 2000]. However, digital signal processing is based on regular sampling, so for this kind of application using unit delays would seem appropriate. In particular, digital filter design assumes unit delays.

However, this filter is not particularly useful as it lacks the means to dynamically control the corner frequency.

By employing a few more unit delays in feed-forward and feed-back configurations, and by allowing the the feed-forward and feed-back scaling coefficients to be dynamically controlled in particular ways, it is possible to construct well-behaved, digital filters with a dynamic corner frequency [Smith, 2006a]. A dynamic corner frequency is essential for many classical synthesizer effects, like filter sweeps. In particular, a lot of effort has been invested in capturing the characteristics of classic analogue designs, such as the aforementioned Moog filter [Stilson and Smith, 1996, Huovilainen, 2004].

Here we implement a digital filter, which tries to mimic the Moog transistor ladder filter, originally implemented as an analogue filter.

Let us start with type signature of a filter:

$$\begin{aligned} &\mathit{moogVCF} :: \mathit{SampleRate} \rightarrow \mathit{Frequency} \rightarrow \mathit{Resonance} \\ &\quad \rightarrow \mathit{SF}\ (\mathit{Sample}, \mathit{CV})\ \mathit{Sample} \end{aligned}$$

The filter is parametrized over digital sampling rate, corner frequency and resonance. The latter is a floating point number and should be in the closed interval of $[0, 1]$. If it equals to 0 then no resonance will be observed, otherwise the filter will have resonant character in the region around the corner frequency. Generated signal function receives two signals, the first is an audio signal which should be filtered and the second is a control signal which dynamically changes corner frequency, in a same way as for an oscillator (see Section 3.1).

Moog transistor ladder filter consists of four stages. Our implementation is based on difference equation derived from [Huovilainen, 2004].

$$
\begin{aligned}
y_a(n) &= y_a(n-1) + 2V_t g \left(tanh\left(\frac{x(n) - 4r y_d(n-1)}{2V_t}\right) - tanh\left(\frac{y_a(n-1)}{2V_t}\right)\right) \\
y_b(n) &= y_b(n-1) + 2V_t g \left(tanh\left(\frac{y_a(n)}{2V_t}\right) - tanh\left(\frac{y_b(n-1)}{2V_t}\right)\right) \\
y_c(n) &= y_c(n-1) + 2V_t g \left(tanh\left(\frac{y_b(n)}{2V_t}\right) - tanh\left(\frac{y_c(n-1)}{2V_t}\right)\right) \\
y_d(n) &= y_d(n-1) + 2V_t g \left(tanh\left(\frac{y_c(n)}{2V_t}\right) - tanh\left(\frac{y_d(n-1)}{2V_t}\right)\right)
\end{aligned}
$$

Where $x(n)$ is an input, $y_a(n)$, $y_b(n)$, $y_c(n)$, $y_d(n)$ are outputs of individual stages, r is a resonance, V_t is a constant representing *thermal voltage* of the transistor, g is a function of dynamically modifiable corner frequency (F_c) and digital sampling rate (F_s).

$$g = 1 - e^{-2\pi F_c / F_s} \tag{3.6}$$

Translation of these equations into the Yampa code gives rise to following implementation.

```
moogVCF :: SampleRate → Frequency → Resonanse
   → SF (Sample, CV) Sample
moogVCF sr f0 r = proc (x, cv) → do
  let f = f0 * (2 ** cv)
      g = 1 - exp (-2 * pi * f / fromIntegral sr)
  rec ya ← moogAux⤙ (x - 4 * r * y, g)
      yb ← moogAux⤙ (ya, g)
      yc ← moogAux⤙ (yb, g)
      yd ← moogAux⤙ (yc, g)
        -- 1/2-sample delay for phase compensation
      ye ← iPre 0⤙ yd
      y ← iPre 0⤙ (ye + yd) / 2
  returnA⤙ y
  where
  vt = 40000   -- (2 * Vt) thermal voltage
  moogAux = proc (x, g) → do
    rec let y = ym1 + vt * g * (tanh (x / vt) - tanh (ym1 / vt))
        ym1 ← iPre 0⤙ y
    returnA⤙ y
```

In addition we are introducing a 1/2 sample delay to compensate phase shift, which arises due to unit delay in feedback path. This is to make digital implementation as close to analogue one as possible. For further details see [Huovilainen, 2004].

3.5 Delay Lines

Long delay lines, with delays up to many seconds, are also very useful in signal processing. Obvious applications include implementing artificial reverb and echo [Smith, 2006b]. Another interesting application is for implementing *physical models* of acoustic instruments. Such a model is thus an alternative to the oscillators outlined in Section 3.1. One of the most most famous of these is the Karplus-Strong algorithm for synthesizing plucked strings [Karplus and Strong, 1983]. It simply consists of a delay line and feedback via the simple digital LP filter outlined above. To actually make a sound, the delay lines needs to be excited (or "plucked") with a burst of noise.

Yampa provides a combinator that implements a delay line with a fixed delay:

$$delay :: Time \to a \to SF\ a\ a$$

Using this, the basic Karplus-Strong configuration (without plucking) can be implemented as follows:

$$\begin{array}{l} ks\ t = \mathbf{proc}\ x \to \\ \quad \mathbf{rec} \\ \quad\quad y \leftarrow delay\ t\ 0 \prec x + 0.5 * y + 0.5 * y' \\ \quad\quad y' \leftarrow iPre\ 0 \quad \prec y \\ \quad returnA \prec y \end{array}$$

3.6 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 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:

$$\begin{array}{l} playNote :: NoteNumber \to SF\ a\ Sample \\ playNote\ n = \mathbf{proc}\ \_ \to \mathbf{do} \\ \quad v \quad\quad \leftarrow oscSine\ 5.0 \quad\quad\quad \prec 0.0 \\ \quad s \quad\quad \leftarrow oscSine\ (toFreq\ n) \prec 0.05 * v \\ \quad (e, \_) \leftarrow envBell \quad\quad\quad \prec noEvent \\ \quad returnA \prec e * s \end{array}$$

$$envBell = envGen\ 0.0\ [(0.1, 1.0), (1.5, 0.0)]\ Nothing$$

Figure 3.7 shows a graphical representation of *playNotes*. The auxiliary

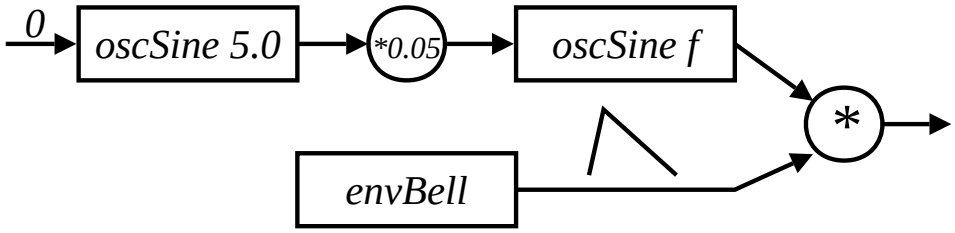


Figure 3.7: Vibrato and bell-like envelope

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.

Finally we can generate a sequence of events to play a C major scale (note number 60 is defined to be "middle C" in the MIDI standard):

```
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)]
⋙ playNotes
```

Chapter 4

A SoundFont-based Polyphonic Synthesizer

In this chapter we implement a SoundFont-based polyphonic synthesizer, which synthesizes and renders MIDI music in real-time. We use synthesizer modules from Chapter 3, and define new ones where necessary. We also exploit Yampa's capabilities to describe systems with highly dynamic structure. The former is used to handle polyphony. Additionally we develop supporting libraries to deal with multimedia file formats and audio input/output.

First of all, we introduce MIDI music, and provide functions to transform a standard MIDI file into the Yampa's event source. Secondly, we implement an interface to SoundFont files and translate instrument descriptions into Yampa's signal functions. Thirdly, we handle polyphony with Yampa's parallel switching combinators. Lastly, we demonstrate how to run defined polyphonic synthesizers and give some performance figures.

4.1 MIDI Music

There are many ways of describing music. Here we focus our attention on MIDI, the most widely used music description standard. It specifies a standard MIDI file format, and a protocol for MIDI message generation and transmission.

The MIDI standard is defined in terms of MIDI messages. Here we give a Haskell data type which serves as a representation of all possible messages. For presentation purposes meta and system exclusive messages are omitted from data type definition, please see the source code for full definition.

```
data Message =
    -- Channel Messages
    NoteOff {channel :: Int, key :: Int, velocity :: Int}
  | NoteOn {channel :: Int, key :: Int, velocity :: Int}
  | KeyPressure {channel :: Int, key :: Int, pressure :: Int}
  | ControlChange {
    channel :: Int
    , controllerNumber :: Int
    , controllerValue :: Int}
  | ProgramChange {channel :: Int, preset :: Int}
  | ChannelPressure {channel :: Int, pressure :: Int}
  | PitchWheel {channel :: Int, pitchWheel :: Int}
    -- Meta Messages
  ...
    -- System Exclusive Messages
  ...
```

Now let us consider some important MIDI messages and there parameters. Parameter *channel* is used to specify the channel number for which the message is intended for. Channels are used to separate musical configurations. For instance, *ProgramChange* message is used to associate different instruments to there corresponding channels.

Parameter *velocity* is used to describe *NoteOn* and *NoteOff* messages. If we consider an instrument with a keyboard, it describes the speed of a key depression or a release. In most cases this parameter is used to control loudness of a played note. Parameter *noteNumber* specifies the ordinal number of the note in a chromatic scale. This is converted to frequency according to the instrument tuning. Usually equal temperaments are used.

PitchWheel message is used to modulate pitch of currently playing notes. *KeyPressure* and *ChannelPressure* messages represent pressure on pressed keys for keyboard based instruments. Their effect depends on an instrument definition.

Events holding MIDI messages serve as an input to our polyphonic synthesizer. If messages are generated from a MIDI keyboard then they will be "sensed" by *reactimate* function and will be fed into a running synthesizer signal function. If MIDI music is available as a standard MIDI file then it is transformed into the Yampa's event source.

A Haskell data type to accommodate contents of standard MIDI files is defined in a supporting library.

```
data Midi = Midi {
  ...
  }
```

For this particular application we do not need to understand the internal structure of MIDI files and the corresponding data structures. A Parser is provided as follows:

$$importFile :: FilePath \rightarrow IO\ (Either\ String\ Midi)$$

It returns *Midi* data structure if file is well formed, otherwise returns informative error message. Finally we provide function which translates a *Midi* data structure into the Yampa's event source

$$midiToEventSource :: Midi \rightarrow SF\ a\ (Event\ [Message])$$

A List is used to encode possibly simultaneous occurrences of MIDI messages.

4.2 SoundFont-based monophonic synthesizer

4.2.1 SoundFont Instrument Description Standard

SoundFont is an open standard describing the behaviour of musical instruments. It standardises sample representation formats, filters, envelopes generators, modulators, and specifies interconnections between them. Another widely used standard is Downloadable Sounds (DLS), standardised by MIDI Manufacturers Association (MMA). Both standards complement MIDI standard with instrument definitions and are intended to be used in conjunction with MIDI protocol.

SoundFont standard defines a file format which contains collections of pre-recorded samples and instrument articulation parameters. Samples are stored in the Resource Interchange File Format (RIFF), where each sample is accompanied by different parameters, most importantly they are tagged with addresses of so called loop points. Loop points are used to define a region of a sample that should be played repeatedly while the note is being sustained. Other important parameters associated with a sample are its native frequency, i.e. the fundamental frequency of the recorded note, and the sampling rate of the sample. Both parameters are necessary to play a sample at different frequencies.

Samples maybe restricted to an interval of note numbers. This means that a sample should be used as the source of an oscillator only for the notes in the specified interval. It is also common to restrict a sample to a velocity interval. Thereby making possible to specify different samples for different velocities. This technique is very useful for some instruments and is called *multisampling*.

An instrument articulation is defined in terms of interconnections of filters, amplifiers, modulators and envelopes, as summarised in Figure 4.1. See [SFS, 2006] for more details about SoundFont synthesis model.

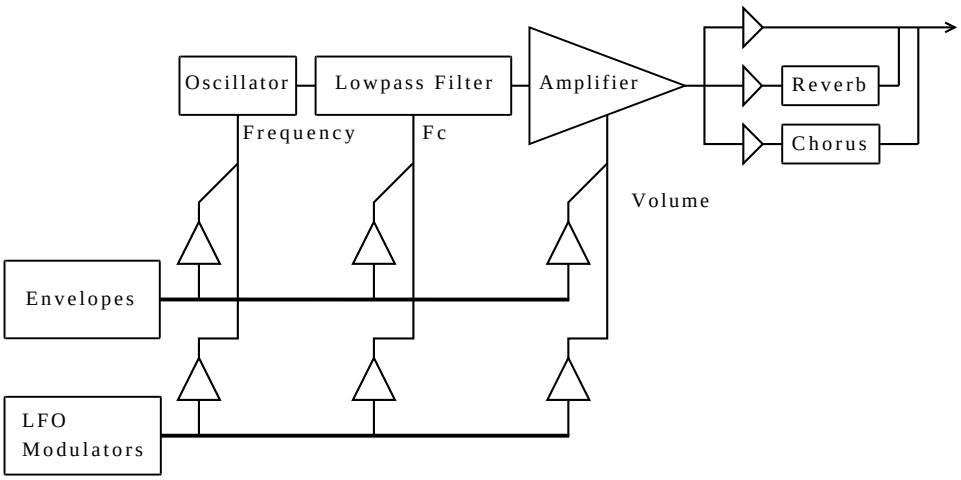


Figure 4.1: The SoundFont synthesis model

Similarity of the synthesis model described in Figure 4.1 and the simple modular synthesizer patch from Section 3.6, is apparent. However, Figure 4.1 is more complex as it is intended to be general definition of various instruments.

Construction of a monophonic synthesizer with articulation facilities is now straight forward. Appropriate signal functions are defined in advance and they are wired together using arrow combinators to obtain final definition of the monophonic synthesizer for the particular instrument. All necessary parameters needed for generating signal functions are imported from an articulation section of a SoundFont file.

4.2.2 Implementing a Sample-based Oscillator

The sample-based oscillator complements the sine oscillator (Section 3.1) in the SoundFont synthesis model (see Figure 4.1). Here is given the implementation of 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. By varying the playback speed, a single sample can be used to play many different notes.

Along with the actual sample data, information on each sample is given, 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 samples is stored in a single array:

type *SampleData* = *UArray Int Sample*

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*.

```
data SampleInfo = SampleInfo {
  smplRate :: SampleRate,
  smplFreq :: Frequency,
  smplStart :: Int,
  smplEnd :: Int,
  smplLoopStart :: Int,
  smplLoopEnd :: Int,
  smplMode :: SampleMode
}
data SampleMode = NoLoop | ContLoop | PressLoop
```

There are three different sample looping modes. *NoLoop* mode indicates that there is no loop region defined in the sample; *ContLoop* mode indicates that there is a loop region and it should be played until the end of the note, i.e. there are no sample points after loop region; *PressLoop* mode indicates that loop region is played only when key is pressed, after key release playback continuous outside the loop region, i.e. there are sample points after loop region.

A sample-playing oscillator can now be defined in much the same way as the sine oscillator from Section 3.1, the main difference being that the periodic function is now given by array lookup and linear interpolation. Here we give simplified implementation to illustrate the idea. This is why handling of looping modes is omitted.

```
oscSmplSimplified :: Frequency → SampleData → SampleInfo
  → SF CV Sample
oscSmplSimplified 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.

4.2.3 Combining the Pieces

Given a sample-based oscillator, a complete SoundFont synthesizer can be obtained by wiring together the appropriate modules according to the SoundFont synthesis model shown in Figure 4.1, just like the simple monophonic synthesizer was constructed in Section 3.6.

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.

$$\begin{aligned}&\textbf{type } \mathit{MonoSynth} = \mathit{Channel} \to \mathit{NoteNumber} \to \mathit{Velocity}\\ &\quad \to \mathit{SF}\ (\mathit{Event}\ [\mathit{Message}])\ (\mathit{Sample}, \mathit{Event}\ ())\end{aligned}$$

The output event indicates that the release phase has been completed and the playing of a note thus is complete. This event is very handy when implementing a polyphonic synthesizer with multiple monophonic synthesizers running in parallel, as will be described in section 4.3.

Overall, collection of monophonic synthesizers indexed by instrument identification number and bank number is obtained from SoundFont file. In MIDI terms, the instrument identification number is the same as the *ProgramNumber*, whereas the bank number is introduced to group instruments according to their properties. For example it is common that bank number zero is used for melodic instruments and 128 for percussive instruments. As the *ProgramNumber* is limited to 127 according to the MIDI standard, the bank number allows many more instruments to be defined in a single SoundFont file.

Finally we can write the definition of an instrument collection like this:

$$\begin{aligned}&\textbf{type } \mathit{Orchestra} = \mathit{Map}\ (\mathit{Bank}, \mathit{ProgramNumber})\ \mathit{MonoSynth}\\ &\textbf{type } \mathit{Bank} = \mathit{Int}\\ &\textbf{type } \mathit{ProgramNumber} = \mathit{Int}\end{aligned}$$

4.3 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 MIDI music.

4.3.1 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 [Nilsson et al., 2002] 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))   -- Sig. func. to switch into
  → SF a (col b)
```

The first argument is an initial collection. The second argument is a signal function which observes external input signal and output signals from the collection in order to determine when switching should occur. The third argument is the function which is invoked when the switching event occurs. It returns a new signal function to switch into based on the collection of signal functions previously running and the value carried by the switching event. This allows collection to be updated and then switched back in. This is often done by using *pSwitchB* again. More detailed descriptions of Yampa's switching combinators are provided in [Nilsson et al., 2002].

pSwitchB broadcasts input to each signal function in the maintained collection. For our application, there is no need for more advanced routing functionality, because each monophonic synthesizer "knows" to which MIDI events it should respond, discarding irrelevant messages.

The combinator *pSwitchB* is similar to *switch* described in Secttion 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 synthesizer termination events, while *preserving* the state of all other signal functions in the collection.

The overall structure of the polyphonic synthesizer is shown in Figure 4.2.

Signal function *triggerChange* generates a switching event when reconfiguration is necessary. The function *performChange* computes a new collection of synthesizer 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

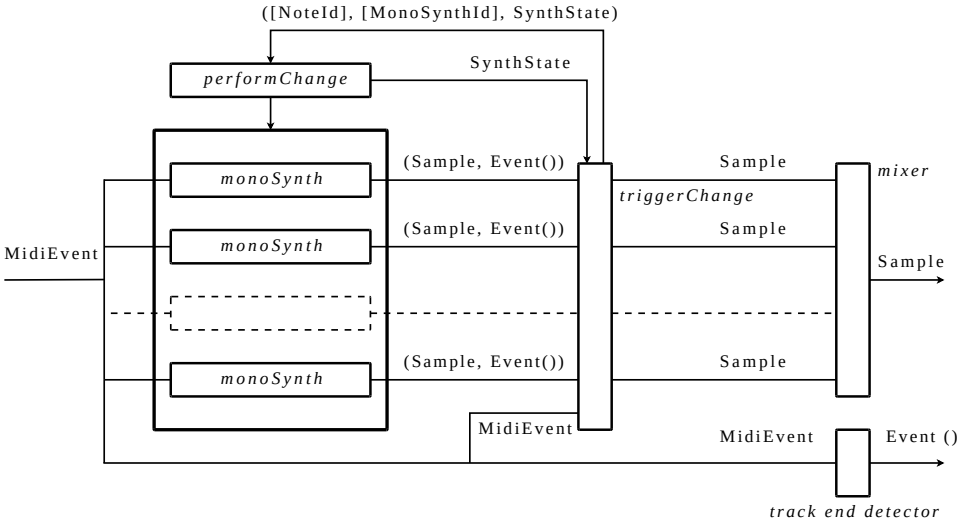


Figure 4.2: Overall structure of the polyphonic synthesizer

synthesizer instance. This can be seen as a collection of audio signals. The signal function *mixer* sums them into a single audio signal.

Finally we give type signature of polyphonic synthesizer

$$\textbf{type}\ \mathit{Synth} = \mathit{SF}\ (\mathit{Event}\ [\mathit{Message}])\ (\mathit{Sample}, \mathit{Event}\ ())$$

Polyphonic synthesizer is a signal function which receives discrete events holding MIDI messages and outputs audio samples together with an event indicating that the synthesizer received the *TrackEnd*[1] message and that it is time to finish synthesis.

4.3.2 Executing Synthesizers

In this section we illustrate three functions to execute (or "to reactimate") already implemented synthesizer instances.

First of all we provide a function which executes synthesizer signal function and writes the result into the WAV audio file.

```
synthesizeWav ::
  FilePath   -- path to audio file
  → SampleRate
  → SF a (Sample, Event ())
    -- signal function which generates audio and termination event
  → IO ()
```

[1] *TrackEnd* is a MIDI meta message and indicates that music is over and synthesizer can be switched off.

Here is an example of its use. If we obtain *synth* :: *Synth* and *midi* :: *Midi* from SoundFont and MIDI files we can execute *synthesizeWav* in the following way:

$$synthesizeWav\ \texttt{"result.wav"}\ 44100\ (midiToEventSource\ midi \ggg synth)$$

Secondly, we provide function which synthesizes and renders audio directly to sound-card in real-time:

$$synthesize :: SampleRate \rightarrow SF\ a\ (Sample, Event\ ()) \rightarrow IO\ ()$$

This can be used in a similar way as *synthesizeWav*, to play MIDI files in real-time, for example.

Lastly, we provide function which, given MIDI music synthesizer, executes a simple graphical user interface to interact with a user and synthesize audio in real-time:

$$synthesizeInteractive :: SampleRate \rightarrow Synth \rightarrow IO\ ()$$

Most importantly, it monitors keyboard events and translates them into MIDI messages, which are than fed into the running synthesizer instance.

All functions described in this section are just convenient wrapper functions, internally implemented using Yampa's *reactimate* function (see Section 2.6).

4.3.3 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 6.8, a moderately complex score like Mozart's *Rondo Alla Turca*, can be rendered as fast as it plays at a 22 kHz sampling rate using a SoundFont[2] piano definition. When connected to a MIDI keyboard to synthesize audio in real-time, on the same sampling rate, as many as eight keys can be pressed simultaneously without any interruption in the synthesized sound.

4.4 Supporting Libraries

In this section we illustrate a supporting library which may also be useful outside the scope of this project. It is entirely implemented in standard Haskell. We provide functions to efficiently read, write and manipulate MIDI, SoundFont and WAV multimedia file formats.

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

Haskell data types have been defined to represent the contents of multimedia files, and efficient parsers and builders were implemented, enabling reading and writing to file, and binary serialization.

Here we list some of the top level functions, qualified by their respective module names.

$$Codec.Midi.importFile :: FilePath \rightarrow IO\ (Either\ String\ Midi)$$
$$Codec.Midi.exportFile :: FilePath \rightarrow Midi \rightarrow IO\ ()$$

$$Codec.Wav.importFile :: FilePath \rightarrow IO\ (Either\ String\ Wav)$$
$$Codec.Wav.exportFile :: FilePath \rightarrow Wav \rightarrow IO\ ()$$

$$Codec.SoundFont.importFile :: FilePath \rightarrow IO\ (Either\ String\ SoundFont)$$
$$Codec.SoundFont.exportFile :: FilePath \rightarrow SoundFont \rightarrow IO\ ()$$

Correctness of library functions has been validated with QuickCheck [Claessen and Hughes, 2000] and Haskell Program Coverage (HPC) tool-kit [Gill and Runciman, 2007]. Please see the source code for further details.

Chapter 5

Related Work

Haskore [Hudak et al., 1996] 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", realizing all manners of algorithmic composition algorithms, for example. Haskore itself does not deal with defining instruments, but see the discussion of HasSound below. Describing musical scores or algorithmic composition 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, and that the score writing abstractions need not be unduly influenced by traditional notation or technical limitations of the MIDI standard (which is very keyboard centric). 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 [Vercoe, 2007]. Fundamentally, it is a modular synthesizer, enabling the user to connect predefined modules in any conceivable manner. Originally it was intended for batch processing, processing a score specified in its propriety score-writing notation, and recording the result as audio for subsequent playback. However, given a sufficiently powerful computer, and not too demanding instrument definitions, it can also be used for real-time performance. 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. Literally every synthesis method invented to date is supported, and it provides very sophisticated implementations of various sound processing algorithms (filters, effects etc.), guaran-

teeing first-class sound quality. Obviously, what we have done in this report 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.

As to performance, we have not compared with Csound. There is little doubt that our implementation as it stands is much less efficient. However efficiency was not our primary concern: this is a proof of concept only. That said, the implementation is not so slow so as to be unusable. Simple MIDI files can be translated as fast as they are played with acceptable audio quality.

HasSound [Hudak et al., 2005] 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, and in some respects in a better way than the standard Csound language. The reason is that the HasSound design, besides offering the usual benefits of working in Haskell, hides some imperative aspects of Csound (the low-level details are handled by the HasSound compilation phase). On the other hand, the end 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. But for many applications, the practical advantages of having access to something like Csound, is likely to to be more important.

Low-level audio processing and sound-generation in Haskell has also been done earlier. For example, Thielemann [Thielemann, 2004] 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. Construction of musical instruments isn't considered, and the oscillators and filters that are considered are too basic for synthesis work as they cannot be modulated.

Karczmarczuk [Karczmarczuk, 2005] presents a framework with goals similar to ours using a stream-based signal representation. Like Thielemann's work, this work is based on representing signals by co-recursively defined streams. Karczmarczuk chose to work in the lazy language Clean as opposed to Haskell, but for the purposes at hand these languages are so similar that this does not matter much. Karczmarczuk focuses on musically relevant algorithms and presents a number of concise realizations of physical instrument simulations, including the Karplus-Strong model of a plucked string [Karplus and Strong, 1983], reverb, and filters. He also presents an efficient, delay-based sine oscillator, (no need to use the sine function as

such: the definition oscillates because the oscillating signal is the solution to the defining co-recursive difference equation), and does consider how to modulate its frequency by another signal to create vibrato. Finally, he also considers delay lines, including *dynamic* delay lines with fractional delays, something which Yampa currently lacks, but which is needed for many musical effects, such as chorus, phasors, and rotating speaker emulations.

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[1], 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.

[1]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.

Chapter 6

Conclusions

FRP and Yampa address application domains that have not been traditionally associated with pure declarative programming. For example, in earlier work Yampa was applied to video game implementation [Courtney et al., 2003], and others have since taken those ideas much further [Cheong, 2005]. 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 [Nilsson et al., 2002] 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.

Bibliography

[SFS, 2006] (2006). *SoundFont Technical Specification*. Version 2.04.

[Cheong, 2005] Cheong, M. H. (2005). Functional programming and 3D games. BEng thesis, University of New South Wales, Sydney, Australia.

[Claessen and Hughes, 2000] Claessen, K. and Hughes, J. (2000). Quickcheck: a lightweight tool for random testing of haskell programs. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279, New York, NY, USA. ACM.

[Courtney et al., 2003] Courtney, A., Nilsson, H., and Peterson, J. (2003). The Yampa arcade. In *Proceedings of the 2003 ACM SIGPLAN Haskell Workshop (Haskell'03)*, pages 7–18, Uppsala, Sweden. ACM Press.

[Gill and Runciman, 2007] Gill, A. and Runciman, C. (2007). Haskell Program Coverage. In *Proceedings of the 2007 ACM SIGPLAN Workshop on Haskell*. ACM Press.

[Giorgidze and Nilsson, 2008] Giorgidze, G. and Nilsson, H. (2008). Switched-on yampa. In *Practical Aspects of Declarative Languages*, volume Volume 4902/2008, pages 282–298. Springer Berlin / Heidelberg.

[Hudak et al., 2003] Hudak, P., Courtney, A., Nilsson, H., and Peterson, J. (2003). Arrows, robots, and functional reactive programming. In Jeuring, J. and Peyton Jones, S., editors, *Advanced Functional Programming, 4th International School 2002*, volume 2638 of *Lecture Notes in Computer Science*, pages 159–187. Springer-Verlag.

[Hudak et al., 1996] Hudak, P., Makucevich, T., Gadde, S., and Whong, B. (1996). Haskore music notation - an algebra of music. *Journal of Functional Programming*, 6(3):465–483.

[Hudak et al., 2005] Hudak, P., Zamec, M., and Eisenstat, S. (2005). HasSound: Generating musical instrument sounds in Haskell. NEPLS talk, Brown University. Slides: `http://plucky.cs.yale.edu/cs431/-HasSoundNEPLS-10-05.ppt`.

[Hughes, 2000] Hughes, J. (2000). Generalising monads to arrows. *Science of Computer Programming*, 37:67–111.

[Huovilainen, 2004] Huovilainen, A. (2004). Nonlinear digital implementation of the moog ladder filter. In *Proceedings of the 7th International Conference on Digital Audio Effects (DAFx'04)*, pages 61–64, Naples, Italy.

[Karczmarczuk, 2005] Karczmarczuk, J. (2005). Functional framework for sound synthesis. In *Seventh International Symposium on Practical Aspects of Declarative Programming*, volume 3350 of *Lecture Notes in Computer Science*, pages 7–21, Long Beach, California. Springer-Verlag.

[Karplus and Strong, 1983] Karplus, K. and Strong, A. (1983). Digital synthesis of plucked string and drum timbres. *Computer Music Journal*, 7(2):43–55.

[Nilsson et al., 2002] Nilsson, H., Courtney, A., and Peterson, J. (2002). Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop (Haskell'02)*, pages 51–64, Pittsburgh, Pennsylvania, USA. ACM Press.

[Paterson, 2001] Paterson, R. (2001). A new notation for arrows. In *Proceedings of the 2001 ACM SIGPLAN International Conference on Functional Programming*, pages 229–240, Firenze, Italy.

[Sculthorpe, 2007] Sculthorpe, N. (2007). Efficient, scalable implementation of functional reactive programming. Phd first year report, University of Nottingham, Nottingham, UK.

[Smith, 2006a] Smith, J. O. (2006a). *Introduction to Digital Filters, August 2006 Edition*. CCRMA. http://ccrma.stanford.edu/~jos/filters06/.

[Smith, 2006b] Smith, J. O. (2006b). *Physical Audio Signal Processing: for Virtual Musical Instruments and Digital Audio Effects, August 2006 Edition*. CCRMA. http://ccrma.stanford.edu/~jos/pasp06/.

[Stilson and Smith, 1996] Stilson, T. and Smith, J. O. (1996). Analyzing the Moog VCF with considerations for digital implementation. In *Proceedings of the 1996 International Computer Music Conference*, Hong Kong. Computer Music Association.

[Thielemann, 2004] Thielemann, H. (2004). Audio processing using Haskell. In *Proceedings of the 7th International Conference on Digital Audio Effects (DAFx'04)*, pages 201–206, Naples.

[Vercoe, 2007] Vercoe, B. (2007). *The Canonical Csound Reference Manual*. MIT Media Lab.

[Wan and Hudak, 2000] Wan, Z. and Hudak, P. (2000). Functional reactive programming from first principles. In *Proceedings of PLDI'01: Symposium on Programming Language Design and Implementation*, pages 242–252.