Musikell : A Functional Reactive Approach to Sound Synthesis

Ian Fitzpatrick, B.A (Mod)

Master in Computer Science (MCS)
University of Dublin, Trinity College
Supervisor: Dr. Glenn Strong
Submitted to the University of Dublin, Trinity College, May, 2013

Declaration

I, Ian Fitzpatrick, declare that the following dissertation, except where otherwise stated, is entirely my own work; that it has not previously been submitted as an exercise for a degree, either in Trinity College Dublin, or in any other University; and that the library may lend or copy it or any part thereof on request.

Signature

Date

Attestation

I understand the nature of plagiarism, and I am aware of the Universitys policy on this. I certify that this dissertation reports original work by me during my University project except for the following :
The images used representing Yampa Combinators in Chapter two are taken from the Haskell Wiki `{(http://www.haskell.org/haskellwiki/Yampa)`.
Images used in Chapter three are taken from the website;
`http://beausievers.com/synth/synthbasics/`.

Signature

Date

Summary

Software sound synthesis is an active area in the programming community. Many different "soft synths" have been created using a variety of languages. This dissertation outlines the work that we carried in order to create a software synthesiser framework using the programming language Haskell, and the functional reactive library Reactive-banana.

Chapter one, provides a broad overview of the dissertation, giving some brief background to the area of sound synthesis, it then outlines the goals of this dissertation.

The second chapter gives a detailed background into Reactive Programming, Functional Programming and Functional Reactive Programming, before taking a closer ooh at Reactive-banana.

Chapter three, outlines the decisions we made and how they influenced the design of our framework, including the generation of samples, and the inner state of the framework.

The penultimate chapter, discusses the use of Functional Reactive Programming, in particular the appropriateness of Reactive-banana as applied to this domain. We also discuss the strengths and shortcomings of our framework.

Finally, in chapter five we review what has been discussed and provides future work which could be undertaken.

Acknowledgements

This dissertation would not be possible without the support and encouragement of many people. Firstly I would like to thank Dr. Glenn Strong, for his assistance and guidance throughout this dissertation.

I would also like to thank my friends and classmates, who kept me going with constant support and motivation.

Finally, I would like to thank my family, in particular my parents, without them none of this would have ever been possible.

Contents

List of Figures

Chapter 1

Introduction

1.1 Dissertation Outline

Chapter one of this dissertation talks briefly about sound synthesisers and functional programming. Chapter two discusses the ideas of Reactive Programming, Functional Programming, and Functional Reactive Programing, before discussing in more detail the FRP library Reactive-banana. Chapter three presents our approach to designing our sound synthesiser, giving motivations and code to how we achieved these results. Chapter four discusses, Functional Reactive Programming as a viable programming paradigm, and whether Reactive-banana was a suitable library for the task of sound synthesis. The final chapter summarises our work and outlines some future work which could be carried out.

1.2 Background

Sound Synthesis is the process of creating sounds using electronic hardware or software, from scratch. Historically specialised hardware synthesisers have dominated this field, however due to recent advances in technology, it is possible for a person to digitally synthesise sound on their PC hardware using new software.

Software synthesisers have been designed in many different programming languages, one such "softsynth" named "Pure Data" which is built using Python, abstracted away from the classical idea of dials and sliders, to a create a more programmatic feel for music synthesis.

Haskell is a functional programming language that is used in many different business and programs around the world. Haskell allows the user to take a high level approach to programming, allowing an increased focus on writing neat and maintainable code.

There have been several examples of sound synthesis software built using Haskell such as Haskore which focuses on describing sound in terms of sequences of notes[14] or HasSound which is essentially an interface to CSound[13].

Functional Reactive Programming is a method of modeling reactive behaviors in purely functional languages, such as Haskell. In purely functional programming languages there are no side-effects, however there are cases where side-effects are necessary, such as with user interaction. Functional Reactive Programming aims to meet the needs of programmers who wish to model these cases without introducing the side-effects associated with them.

There have also been cases of synthesisers built using the different Functional Reactive libraries, such as Yampa-synth[11]. However these synthesisers dealt with the input of Midi-files and the manipulation of such files.

1.3 Aim

The aim of this dissertation was to explore the possibility of creating a sound synthesis framework (christened Musikell) that allows a user to programmatically create sounds through the specification of different functions, using the programming language Haskell and the Functional Reactive library, Reactive-banana. A follow up to this aim is whether Reactive-banana is a suitable library for such a task, or whether a different approach would have been more feasible. When creating the synthesiser framework, we aim to try and keep the ideas presented as simple as possible, while also keeping the framework self-reliant.

Chapter 2

State of the Art

2.1 Functional Reactive Programming

2.1.1 Reactive Programming

What is Reactive Programming?

Normally we can consider a program as a function of its inputs, this is often referred to as transformational [7]. However there are other approaches that aim to model a continuous flow of reactions to stimuli which occur throughout a programs life. Programs modeled like so are said to react to their environments.

Spreadsheets

An appropriate analogy for reactive programming is that of a spreadsheet program, such as Microsoft's Excel program. If we were to consider there to be two types of box that we can make use of. The first type is what we shall call a formula box, in it there is contained some information about how it is related to another box. The second type of box will just be an input box, where we can add any type of text.

In the traditional transformational approach to programming, when we enter new values into the input boxes, nothing will happen, we would have to conceive a third type of operation, such as a refresh button that would force the program to check the different input boxes and update the various other boxes as required. From a programming point of view, we explicitly have to create the relationships between values and how they will update as a program runs.

In reactive programming however, things operate in a more dynamic manner. As we enter values into the input boxes the formula boxes that are related to this box will

update automatically, in fact if there are formula boxes related to other formula boxes then those will also be updated. In a set large enough we could in fact see the data propagate through the number of formula boxes till it reached a point where all related boxes were up to date. The relationships within the spreadsheet are implicitly defined as part of the formulae boxes, leaving the programmer free to reason with "what" the boxes are doing, as opposed to how they would have to deal with "how" they do it.

Why Reactive Programming?

The benefit of using reactive programming is that once we have set up the relationships within a program, we need only worry about entering our own data and the program will handle the rest, pushing the changes throughout. As mentioned, Reactive Programming aims to free the programmer from lower-level concerns and decisions by providing a greater abstraction, allowing programmers to reason about what a program does and focus less on how it does it. The most obvious application of reactive programming is when programming Graphical User Interfaces, we can set relationships to different points of the interface and allowing the user to change one portion of the interface changes another. A second reason to use reactive programming is that it removes the necessity to keep track of multiple different variables that are related, for example;

```
a = 10
b = a + 1
a = 12
```

In imperative programming we would need to keep track of wherever the 'a' was updated so that we could update the 'b'. In reactive programming we could imagine that we have defined a relationship between 'a' and 'b' removing the onus from the programmer to keep track of 'b'. As 'a' varies so indeed does 'b'.

2.1.2 Functional Programming

2.1.3 Introduction to Haskell

Functional programming is a programming paradigm, one that aims to model computations as the evaluations of expressions, as opposed to an imperative programming paradigm that models how each instruction changes the state of a program. Functional programming requires that all functions are first class citizens, that is a function can be passed to, and returned from another function. Haskell is a purely functional program-

ming language, which has a strictly typed system. Apart from having the benefits that come with abiding by the functional paradigm. Haskell employs a powerful type system that ensures once the program compiles that there will be no run time errors, and only bugs to do with the logic of the program itself.

Another benefit of Haskell is that it is a very high level language, that allows users to write neat and maintainable code, that is often very short and easy to reason with.

Another useful feature of Haskell, is that a programmer has two ways in which to evaluate an expression; strictly or lazily. Strict evaluation as the name would imply is done as soon as the expression is encountered, imperative languages such as Java and C++, are intrinsically strict. Lazy evaluation of an expression however is done only when the result of an expression is needed by some other larger expression. We can think of lazy evaluation as keeping a todo list and only crossing off items on the list once they've become a necessity[1].

A discussion on Functors and Applicative Functors, slightly less well known but important features of Functional Programming follows.

Functors

Within Haskell there resides many different typeclasses that one can make a new datatype an instance of. For example, by making something an instance of the Eq typeclass we can then equate the same datatypes.

Functor is another typeclass, we make some type an instance of Functor when we want to be able to map over that type[15]. Functors are defined in the following way;

```
class Functor f where
      fmap :: (a -> b) -> fa -> fb
```

A concrete example of Functor would be as follows;

```
instance Functor Maybe where
      fmap f (Just x) = Just (f x)
      fmap f Nothing = Nothing
```

As we can see by being part of the type class Functor we are able to use fmap to alter values within the Maybe type without having to extract the value stored within it, and then wrap the resultant value back up again.

[1]This todo list in Haskell is called a Thunk, more information on Thunks can be found here: http://www.haskell.org/haskellwiki/Thunk

Applicative Functors

Now that we have considered Functors, imagine the case where we wish to map (*) over some Functor, for example, the maybe type;

```
:t fmap (*) Just 3
> fmap (*) Just 3 :: Maybe (Int -> Int)
```

We get back a function wrapped in the Maybe type. Now that we have this, how do we apply the functor containing the function to a functor containing some value. The fact is that with normal Functors we cannot, we could if we desired pattern match over the functor type, for example;

```
apply (Just f) (Just x) = Just (f x)
apply (Just f) Nothing = Nothing
```

However it is preferable to have a more general and abstract way of dealing with this scenario. Applicative Functors are a type class that are designed to do just this [15]. Applicative Functors reside in the Control.Applicative module. Within this module are defined two functions pure and <*>. Much like fmap in the Functor type class there is no default implementation and so we must define our own. First, however let's look at their type signatures.

```
class (Functor f) => Applicative f where
      pure :: a -> f a
      (<*>) :: f (a -> b) -> f a -> f b
```

As we can see to be part of the Applicative type class, a type constructor must be first part of the Functor type class. The first of the two functions - pure - wraps a value in a default context – a minimal context that still yields a value. The <*> function is quite similar in its type signature to the definition of fmap from the Functor typeclass. However instead of taking a regular function, it takes one that is wrapped in a Functor and then applies it to another value that is also wrapped in a Functor, to get the final result.

Finally an example of the two functions in action.

```
> Just (+3) <*> Just 9
Just 12
> pure (+3) <*> Just 10
```

2.1.4 Functional Reactive Programming

Roots

Functional Reactive Programming is a method of modeling reactive behavior in purely functional languages. In a purely functional language, there are no side-effects, however there are cases where side-effects are necessary, such as with user interaction. Functional Reactive Programming arose from the needs of programmers to model these areas without introducing the side-effects associated with them[12].

The first instance of FRP was in the release of Conal Elliott and Paul Hudak's seminal paper "Functional Reactive Animation"[2], in which was described a collection of data types and functions for composing interactive, multi-media animations. In this paper was described for the first time the ideas of Behaviors and Events, where Behaviors are thought of as time-varying values, and Events were described as "arbitrarily complex conditions, carrying possible rich information".

There are several goals that FRP implementations aim to meet[1];

- Safety: Programs should make use of as much of the compilers correctness checking as possible.
- Efficiency: As most FRP programs will be expected to respond in real time, efficient operation and optimization are necessary
- Composability: FRP should maintain the ability to allow larger programs to be built from smaller programs

Since "Functional Reactive Animation", the concepts presented have stayed similar, in that there are still Behaviors (Signals in some implementations) and Events. FRP achieves reactivity through modeling how Behaviors are updated in response to events that occur.

Numerous semantic models have emerged since the first paper, as much of the difficult work involved with FRP is the definition of a suitable semantic model. Two of the main models that arose are "Classic FRP" which is closely based off Conal Elliott's work. Classic FRP treats Behaviors and Events as first class citizens in the language, and

[2]Although the term Functional Reactive Programming was never used in the paper, it surfaced in subsequent works

allows them to be directly manipulated by the language constructs. Signal function semantics on the other hand, deal with the concept of Signals, but do not allow them to be acted upon in the same way as Behaviors, rather functions on Signals are manipulated and made reactive.

Libraries

There are many different FRP libraries. Each tries to express how they feel is the best way to implement the semantics first proposed by Conal Elliott. There are several different FRP libraries [3],the following section outlines what we see as the current main contributors to FRP based programming.

Fran

Fran, although now considered outdated was the first library to properly implement the ideas of Behaviors and Events as used in more modern FRP libraries, therefore it is remiss to discuss later libraries without discussing to to a small degree this library.

Fran arose from the creators' belief that there did not exist a sufficiently high-level abstraction for dealing with the construction of interactive multimedia animation. Mainly the difficultly lay with the fact that programs had to explicitly manage common implementation duties, such as stepping forward discretely in time, and capturing and handling sequences of motion input events. The authors believed that by allowing programmers to express the "what" of an animation, one could hope to then automate the "how".

Fran captures the essence of modeling through four different concepts[10].

1. Temporal Modeling Values called Behaviors which vary over time are the main values of interest. They are first class values and are built up compositionally. An example is the expression;

   ```
   bigger (sin time) circle
   ```

   At time t, the circle has size sin t. Allowing the circle to change size naturally as time progresses.

2. Events, like Behaviors, are first-class values. Events can refer to happenings in the real world, such as mouse button presses, but also to predicates based on animation

[3] A list is available at: http://www.haskell.org/haskellwiki/Functional_Reactive_Programming#Libraries

parameters, for example the collision of two items in a game. Events can be combined together to an arbitrary degree of complexity, for example we could model the event describing the first left button press after time `t0` is simply `lbp t0`, another describing time squared being equal to 5 is just;

```
predicate (time^2 == 5) t0
```

We can then combine these two events using the `.|.` operator;

```
lbp t0 .|. predicate (time^2 == 5) t0
```

3. Declarative Reactivity, many Behaviors are naturally expressed in terms of reactions to events. These reactive Behaviors have declarative semantics in terms of temporal compositionally rather than an imperative semantics in terms of state changes. For example, a piece of code that alternates between red and blue on each mouse button click :

```
colorCycle t0 =
   red 'untilB' lbp t0 *=> \ t1 ->
   blue 'untilB' lpb t1 *=> \ t2 ->
   colorCycle t2
```

4. Polymorphic Media, the variety of time-varying media and parameters have their own type-specific operations but they fit into a common framework of Behaviors and reactivity.

Without going into more detail than is necessary about the underlying semantics of Fran, I have shown the basic ideas that Fran initially "brought to the table" and the groundwork that it lay for future Functional Reactive Libraries to build upon. In the following sections we will highlight how current Functional Reactive libraries have built and improved upon these ideas.

Yampa

Yampa is a language embedded in Haskell for describing Reactive Systems[9]. It was created by Paul Hudak and like all current FRP libraries is based on the ideas from Fran.

Yampa has two central concepts Signals and Signal Functions. A signal is a function from Time to to a value. We can think of the following definitions as the theoretical way that these functions are implemented, in actual implementations the library has some changes.

```
Signal a = Time -> a
```

Time is continuous and is represented by a non-negative real number[4]

A Signal Function on the other hand is a function that operates over signals;

```
SF a b = Signal a -> Signal b
```

Signal Functions are first class citizens in Yampa, Signals however are not. It is easiest to think of Signal and Signal Functions using a flow chart analogy, where wires represent Signals and boxes represent Signal Functions with one type of Signal entering a box and another type exiting.

Yampa's Signal Functions are an instance of the arrow framework proposed by Hughes. Two of the main combinators are;

```
arr :: (a -> b) -> SF a b
```

and

```
(>>>) :: SF a b -> SF b c -> SF a c
```

Graphically these functions can be expressed as such.

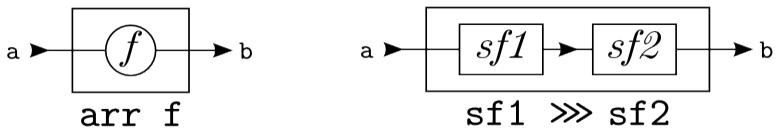

Figure 2.1: A graphical representation of Yampa Combinators, as we can see the `arr` function lifts a regular function into a Signal Function, and the >>> function can compose multiple Signal Functions together.

Finally although we can model some aspects of programming as continuous signals, other sources such as a mouse button being pressed are more naturally described as dis-

[4]In current implementations of Yampa, Time is a synonym of Haskell's Double type

crete events. As such Yampa provides another type to work with;

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

A Signal value whose output signal is of type (Event t) for some type t is called an Event Source. The value carried by an event occurrence may contain some information about the occurrence.

Yampa is currently one of the more popular FRP libraries and users have implemented several different games, music synthesisers and robotics libraries using it. Its strength lies in the fact that the arrow framework helps to avoid a time leak [5] a problem that affected earlier FRP libraries, the arrow syntax also allows a natural modeling of flow between the different modules that make up a program.

2.1.5 Reactive-banana

Unlike other FRP libraries, Reactive-banana takes a more applied approach to Functional Reactive Programming, with the creator Heinrich Apfelmus focusing on building the system before publishing his work. One of the main motivations behind the creation of Reactive-banana was the simplification of coding graphical user interfaces with Functional Reactive Programming. As such the library can be hooked into any existing event based library such as wxHaskell. Reactive-banana can be classed as being of the "classic" FRP style, as semantically the variations in Reactive-banana and Conal Elliott's Fran are negligible. However, the implementations of the two are very different (as will be discussed further on), this gives Reactive-banana greater efficiency. This efficiency along with the ability to easily swap between functional reactive and pure functional programming, whilst coding are the main reason we have chosen to implement the sound synthesiser in Reactive-banana.

What follows is an overview of some of the key ideas of Reactive-banana.

Behaviors

Like Fran, Reactive-banana uses the idea of Behaviors to model time-varying values, semantically we can think of Behaviors like so;

```
type Behavior t a = Time -> a
```

[5] Time leaks will be discussed in chapter 4

Behaviors are instances of the Functor and Applicative Functor typeclasses so we are able to map over a Behavior as we would with other types. Accompanying Behaviors are several helper functions that are quite useful, some are mentioned further down.

Events

Events are also once again used to model a sequence of event occurrences, syntactically we can think of them as;

```
type Event t a = [(Time, Event)]
```

Like Behaviors, Events are instances of the typeclass Functor. However they are not part of the Applicative Functor typeclass. Events are used in Reactive-banana to model discrete occurrences of something, for example the button being pressed on a GUI. You will remember that this is the same as how were they described in Fran, a difference however lies in the implementation of this idea.

Where Fran models Events as something like;

```
type Event a = [(Time, a)]
```

Reactive-banana's implementation is closer to this;

```
type Event a = [(Time, Maybe a)]
```

This may seem like a small change, but this becomes important when using the combinator function "union" (discussed below), in Fran's implementation we have to wait for an occurrence of both Events that we are uniting so we can pattern match on them both before we can proceed, whereas with the Reactive-banana implementation by introducing a condition that Events are "synchronized" in the sense that Events indicate their occurrence at that time [5]. If an event does not occur at this time then it is Nothing, so it can then be implemented as follows;

```
union ((t1,x1):e1) ((t2,x2):e2) = (t1, combine x1 x2) : union
   e1 e2
      where
            combine (Just x) Nothing = Just x
            combine Nothing (Just y) = Just y
```

```
combine (Just x) (Just y) = Just x
combine Nothing Nothing = Nothing
```

This way we can always ensure an event fires correctly in the stream at a time. Fran has a way around this case by using a function called "unamb" to help choose the correct event when it occurs, however it is an extremely inefficient in its implementation, requiring the use of two threads.

Combinators

Reactive-banana, like Yampa gives us several different combinators to provide extra abstraction, and allow for an increase in efficiency. Some of the key combinators in Reactive-banana are[6];

```
stepper :: a ->Event t a -> Behavior t a
```

This function allows the programmer to essentially store the event that is triggered in a Behavior for future use. The value will continuously update as new events arrive.

```
accumB :: a -> Event t a (a -> a) -> Behavior t a
```

This function is similar to stepper, however instead of simply replacing the initial value with one from the event, the event carries a function that can operate over the value. The result of that function is then stored in the Behavior.

```
apply :: Behavior t (a -> b) -> Event t a -> Event t b
```

Apply takes time-varying function and applies it to a stream event, note due to Haskell's purity , what we are left with is two streams whenever an event is fired in the first stream, a transformed event is fired in the second stream.

```
union :: Event t a -> Event t a-> Event t a
```

This is the function that we discussed earlier. This function combines two different events streams that are of the same type into one unified stream. This is useful for when

[6] A full list of combintors can be found at http://hackage.haskell.org/packages/archive/reactive-banana/0.7.0.1/doc/html/Reactive-Banana-Combinators.html

we wish to have two different events affect one Behavior.

Reactive-banana Example

Now that we have looked at some of the functions of Reactive-banana, it is best to provide a concrete example. What follows is a section of code from Reactive-banana's `counter.hs` code[7].

```
let networkDescription :: forall t. Frameworks t
      => Moment t ()
      networkDescription = do

      eup  <- event0 bup command
      edown <- event0 bdown command
      let
            counter :: Behavior t Int
            counter = accumB 0 $ ((+1) <$ eup) `union`
               (subtract 1 <$ edown)
      sink output [text :== show <$> counter]
      network <- compile networkDescription
      actuate network
```

We firstly describe our event network using the `Moment Monad`, which denotes a particular moment in time. In it we describe our inputs and outputs. Our inputs in this case are the Events `eup` and `edown` (which are fired when buttons on the user interface are pressed). The output of this event network is handled by the function `sink` which takes the value from a Behavior and outputs the value onto a graphical user interface.

The Behavior in this program, is the value of the counter. To update the value we use `accumB`, giving the counter an initial starting value of 0. When the Event `eup` is fired, the value stored in counter is increased , and it decreases when the Event `edown` is fired.

Now that that the event network has been described, we compile it into a network to run, and then use `actuate` to start the network.

[7]The entirety of the code can be found at "http://github.com/HeinrichApfelmus/reactive-banana/blob/master/reactive-banana-wx/src/Counter.hs"

2.1.6 Applications

Since its creation FRP has been used in many different areas of programming - such as game design, robotics, and computer vision - to allow an easy modeling between the external interaction of outside stimuli and the internal events of a program. Below are 3 of the more notable endeavors made available to the public.

FRob/YFRob

Functional Robotics (FRob) [17] is an embedded domain specific language for controlling Robots. Initially developed using the Functional Reactive ideas highlighted in Fran. FRob deals with the interaction between a robot and its environment in a purely functional manner. This serves as a basis for composable high level abstractions supporting complex control scheme in a concise way. Although FRob was initially built using the ideas of Fran, new versions see it use Yampa as its main Functional Reactive framework (YFRob).

Frag

Frag is a a 3D game built using Haskell. Its graphics were programmed using HOpenGL, a Haskell binding to the openGL graphics library. More interesting though is that Yampa was used to model both the continuous events of the frames updating and the discrete events associated with the players input. Benchmarking of the game see it perform at a desirable rate, though it was mentioned that improvements could be made through different optimisations [8].

FVision

FVision (pronounced "fission") is a library built in Haskell for the purpose of computer vision. It is based on the C++ library XVision. Using Functional abstractions users of FVision can build new tracking systems quickly and reliably. FVision is again built using Yampa to model a user's input as well as for scanning images [18].

2.2 Synthesisers

2.2.1 Pure Data

Pure Data is an open source visual programming language written in the programing language Python. It aims to allow users to process and generate sound, video, and 2D/3D

graphics. It is available for Windows, Linux and Mac OSX. Pure Data takes several of its ideas from the ways of hardware synthesisers in that we can create several different patches and connect them through different chords. Allowing for different sounds to be generated. Pure Data takes a visual approach to making its patches, functions are represented by objects, which are placed on a screen called a canvas. Pure Data serves as one of the main inspirations of the project in that I hope to emulate to some degree the basic sound functionality that Pure Data allows in a Haskell based framework.

2.2.2 Haskell Synthesisers

In Haskell, there have been a variety of different Synthesisers created in various different ways to describe sound in different varieties. What follows is what I see as the main synthesisers in Haskell.

HasSound

HasSound is a Haskell frontend to CSound. HasSound definitions are compiled to Csound specifications. As such HasSound has ready access to the plethora of Csound features, however the drawback to this is that one cannot do anything in HasSound that cannot be done Csound. One of the main ideas of my dissertation is to build a framework that relies mainly on Haskell, as such Csound is not a particularly good base for this project.

Yampa-Synth

This was a modular synthesiser designed by George Giorgidze and Henrik Nilsson for their paper Switched-On-Yampa [11]. As the name would imply this Synthesiser was created in Yampa which is another Functional Reactive extension for Haskell. The synth served as mostly a proof of concept that Yampa could be used to design a naive reactive system simply and cleanly. However this synth focuses less on a sound generation framework and more on the input of external Midi files and other external inputs such as the key presses of a keyboard to generate its sounds, nonetheless many of the ideas that are presented in this paper serve as cornerstones for design implementations of Musikell.

Reactive-balsa

Reactive-Balsa is a live MIDI event processor that uses the Reactive-banana framework, as such it would amiss to not mention it. Normally one would insert Reactive-balsa between a MIDI input device such as a USB piano keyboard and a MIDI controlled

internal software synthesizer. Again although it uses Reactive-banana, it does not provide a framework to create sound which is the aim of Musikell. Also our framework tries to be as portable as possible, due to Reactive-balsa relying on ALSA - the Advanced Linux Sound Architecture - it will only work on Linux based systems [19].

2.3 Conclusion

In this section, I have laid the groundwork for what one will need to know when programming a Synthesiser using a functional reactive library. The following section will discuss how I implemented my framework in Reactive-banana.

Chapter 3

Design

In the previous chapter we looked at Functional Reactive Programming as a whole, and took a slightly more detailed look at Reactive-banana. In this chapter we are going to discuss how we took these ideas presented and applied them to our Framework.

Our framework is designed to be used as one large network of inputs and outputs. The input of this network can be anything that we choose to connect to the core of the system, such as a keystroke or a button press from the GUI that we have included with this work. The outputs of the network are of course the sounds that have produced over the lifetime of the application.

3.1 Waveforms

While other synthesisers modeled using FRP decided to represent their sound using MIDI-Files [11, 19]we chose a different path and decided to design a much lower level framework that deals with the creation of waves, how a wave evolves over time, and which produces digital samples as output.

3.1.1 Waves Explanation

As one would expect, the theory of musical synthesis is grounded within physics. The best way to understand what is occurring during musical synthesis is to understand the theory of sound.

Sound is made up of pressure waves, these move forward and backwards [6] - not unlike waves in the sea - at a particular frequency. These waves move a persons eardrums with the same frequency. Musical synthesis is the creation of signals which are then turned into sounds waves by a speaker.

When dealing with the synthesiser, we will deal with four basic sound waves; Sine, Sawtooth, Square and Triangle waves.

Sine-waves are the purest of all waves, their name is derived from the fact that they can be plotted using the simple mathematical function Sine. The formula for plotting a Sine-wave is $A \times \sin(wt + \phi)$ where A is the amplitude of the wave, $w =$ the angular frequency, t is the time and ϕ is the phase of the wave.

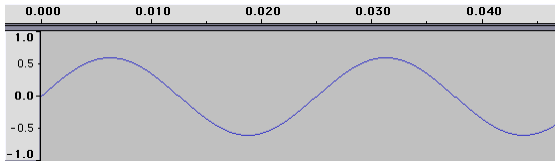

Figure 3.1: A graphical representation of a Sine Wave. Values on the left side of the graph, represent the wave's Amplitude, and the values on top, are time.

Saw-waves are made through the additive synthesis of multiple different Sine-waves. The lowest frequency Sine-wave of a Saw-wave is known as its fundamental frequency. The subsequent Sine-waves are then all integer multiples of this fundamental frequency and they are known as the Harmonics of the Saw-wave. Saw-waves have quite a harsh sound to them.

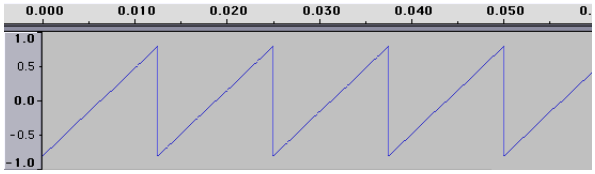

Figure 3.2: A graphical representation of a Saw Wave. Values on the left side of the graph, represent the wave's Amplitude, and the values on top, are time.

Square-waves are similar to Saw-waves, however they only use the odd harmonics of a fundamental frequency.

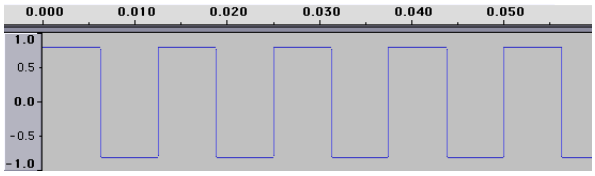

Figure 3.3: A graphical representation of a Square Wave. Values on the left side of the graph, represent the wave's Amplitude, and the values on top, are time.

Finally Triangle-waves are again similar to Saw and Square-waves, they also only use the odd harmonics, however the amplitude of each harmonic drops at a faster rate than that of Square-wave

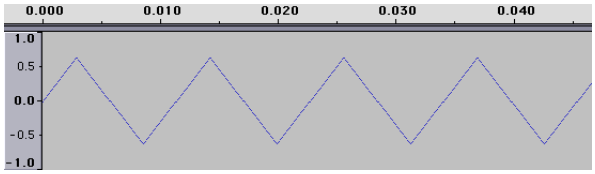

Figure 3.4: A graphical representation of a Triangle Wave. Values on the left side of the graph, represent the wave's Amplitude, and the values on top, are time.

3.1.2 Waves and Behaviors

Due the fact that waveforms vary over time, we decided a natural way to represent them was as a Behavior in our framework. One can consider it as;

```
type SynthState = Behavior Wave
```

Where the type Wave contains type constructors as follows;

```
SinWave Frequency Amplitude Filter
SawWave Frequency Amplitude Harmonics Filter
```

Currently we do not need to worry about the "Filter" part of our types, as it will be explained in a later section. The Behavior above is a very abstract idea of how we should represent waves. When designing the application, we had to decide how to best represent these waves.

One possible way we considered was to generate each value and write it out as we created it, however we felt that this would cause problems as if there was a delay in generating a sound then samples could be delayed, resulting in a skipping sound during playback. An example of an unexpected delay would be if Haskell's garbage collector was to make a pass, it could potentially pause the running of the program, though this would be only for a fraction of a second, it could be enough to disrupt the flow of sound as a user would expect it.

The way we chose to approach this problem instead was to generate an infinite list of points that would represent the overall lifetime of a wave.

```
type SynthState = Behavior [Double]
```

This approach goes part of the way to solving this problem as we now have a large number of values buffered so we do not need to generate them as we need them, providing some relief if a case as pointed out above occurs. The other part of this solution is sampling, which we will now discuss.

3.1.3 Sampling

Now that we have a data type with an infinite list of points that represents a wave, we are presented with two new problems;

- When it comes time for the program to write out to a file, the program will stall as it attempts to write out a never ending stream.
- The second issue is that with an infinite list, there is no proper way to manage the change of a sound over time, by this we mean, if we decided that at first we would play a sine wave and then play a saw wave, with an infinite list we would only have the playback of the saw wave as it would have replaced the infinite list of sine values.

The solution to both these problems is to of course sample points from the infinite list at discrete intervals, and store the values taken until we need to write them out to a file. With this in mind we can present a new definition for our current state.

```
type SynthState = Behavior ([Double], [Double])
```

Where the first list of Doubles, is the samples that we've taken so far, and the second double is the infinite list of points generated.

3.1.4 Sound Generation

Now that we have shown how we wish to represent the waves as values. It is appropriate to show how we generate these values. Oscillators are one of the main components of a synthesiser. The job of an oscillator is to generate a periodic wave. In our framework our implementation is as follows. We have a general function that we use to create our waves;

```
generateWave :: SampleRate -> WaveInfo -> [Double]
```

Before we show the details of `WaveInfo` we can see an example of it in use, the meaning is hopefully clear, if it is not we will be discussing it in full below.

```
generateWave samplePS (WaveInfo s (SinWave f a fil) cv _)
      = (sineWave samplePS s f fil cv)

generateWave samplePS (WaveInfo s (SawWave f a h fil) cv _)
      = (sawWave samplePS s f h fil cv)
```

As we can see it at the heart of it, it takes the wave type that we discussed earlier and passes the information from inside it to a particular function corresponding to each type of wave we can have, as well as some other information.

To understand this information better again let us have a look at the `sineWave` function being called;

```
sineWave :: Int -> Double -> Frequency -> Filter
      -> ControlValue -> SamplesGenerated
sineWave samplePS start freq fil cv
      = map (sine freq cv) $ generateWaves samplePS start

sine f cv t = sin (2*pi*(f*(2^ (fromIntegral cv)))*t)
```

The generateWaves is a simple function that generates a list of Doubles based on a start position and how many samples we wish to produce per second.

```
generateWaves :: Int -> Double -> [Double]
generateWaves samplePS start = [start, step..]
      where
            step = start + 1(fromIntegral samples)
```

More interesting is the `sine` function. As we can see it it just calls the inbuilt `sin` function. We can now see what the purpose of the `cv` value is, it affects the overall frequency of the current wave. We adopted a general convention that increasing this control value by one, doubles the frequency (or increases the pitch by one octave) , and lowering it by one, halves the frequency (or lowers the pitch by one octave).

So now we can discuss the WaveInfo type, it contains extra information important to a wave, but not essential in the generation of one.

```
data WaveInfo = WaveInfo { startPosition :: Double
                         , wave :: Wave
                         , controlValue :: ControlValue
                         , changed :: Bool
                         }
```

The remaining two fields we which we have not yet discussed are the `startPosition` type which as we have seen already, helps generate waves starting at a particular point in the wave cycle. This is important for when we wish to swap waves while they are running, so as to allow the new wave to properly continue from where the previous wave ended. The other field `changed`, is again for when we want to change between waves. It is used to signal to the `gatherSamples` function that once the samples have been gathered that it is to generate a new set of values to begin sampling from. If it is not signaled it will continue to sample from the original "pool" of values.

Now we have shown that we can generate values using the WaveInfo type, it should be obvious at this stage that the values contained within this type are also subject to change over time, for this reason they also need to be part of our main Behavior. So let us once again expand on what we have.

```
type SynthState = Behavior ([Double], [Double], WaveInfo)
```

The final extension to our state that we should consider is the notion of an envelope.

3.1.5 Envelope Generators

What are they?

Envelope generators are another component of a synth, they can be used to control any other component of a synth, normally however they are used to control the amplitude of a signal, to allow for a nice sounding start and finish to the signal[6]. In our implementation we chose to implement just the ability to control the amplitude of the synth. An envelope generator is defined by its levels, and the length of time it should take to reach that level.

Envelope generators normally have four stages to them, these are;

- Attack: This stage ramps the amplitude of a signal up to a specific level, normally full strength.
- Decay: The amplitude drops to a fraction of the full signal.

- Sustain: The amplitude remains all the level decay dropped to, and will remain at that level until a close gate signal occurs.
- Release: The phase after a close gate signal, the amplitude drops off to some number, normally zero.

This has led these type of generators to be known as ADSR generators. Often there is a special case where more stages can be specified, however in our implementation (discussed next) there is no limit to the number stages so we can act as an ADSR generator or a more general one.

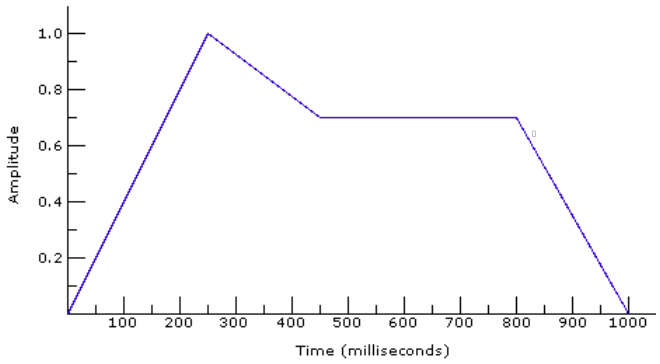

Figure 3.5: An ADSR Envelope Generator. During the attack phase the amplitude of the wave increases to its maximum, before decaying to a fraction of this value, the note will the then be sustained at that level, until it is released..

Implementation

In our implementation, we decided to model envelope generators as another list of Doubles. However it was not convenient for user input to specify a very large list of doubles. To remedy this we instead introduced another new type StepPair;

```
type StepPair = (Time, Level)
```

Both types contained within StepPair are synonymous with the Double type. When programming an interface for Musikell, one can use StepPair to define the levels and time taken to reach these levels for use with an envelope generator.

The next step of course is to convert these StepPairs into a list of Doubles to use. This was done through the following function;

```
beginGenEnvelope' :: Int -> [Double] -> Level -> [StepPair]
   -> [Double]

beginGenEnvelope' samplesPS acc startLevel [] = acc ++
   (repeat startLevel)

beginGenEnvelope' samplesPS acc startLevel ((time,
   nextLevel):xs) = beginGenEnvelope' samplesPS (acc  ++
   [startLevel, sl'..nextLevel]) nextLevel xs
      where
        samples = (time/1000) * (fromIntegral samplesPS)
        sl' | startLevel > nextLevel = (startLevel - 1/samples)
            | otherwise = (startLevel + 1/samples)
```

This function takes the StepPairs provided in the list and using the current level (of the amplitude) decides how to step between each level. To decide on how to step between the levels. The function first takes the time to reach that level (given in Milliseconds), we then convert it to seconds, we then use the samples per second to decide how many samples we need for that number of seconds.

Using the value obtained, we need to decide what the step between the current level and the next level is. We do this by taking the start level and adding one over the number samples (if the next level is above the start level) or subtracting the value (if the next level is below the start level).

Using this step, we generate a list between the start level and the next level where each value is a slight increment to the initial value.

We repeat this process for each StepPair (using the previous pairs, level as the start level) until none are left, we then append an infinite list to the accumulated list, this represent the sustain of the envelope.

The release of the envelope is handled by a separate function, `releaseGenEnvelope'`;

```
endGenEnvelope' :: Int -> Level -> StepPair -> [Double]
endGenEnvelope' samplesPS startLevel (time, nextLevel) =
   [startLevel, sl'..nextLevel] ++ (repeat nextLevel)
```

```
    where
      samples = (time/1000) * (fromIntegral samplesPS)
     sl' = startLevel - (1/samples)
```

This is a much simpler version of `beginGenEnvelope` that simply forces a value down to the specified level. We leave it up to the use to the decide the final level it should stop at.

Once again, as we are constantly changing the value of the envelope generator, it is best to model this as a Behavior aswell. So we can add this into our state;

```
type SynthState = Behavior ([Double] [Double], WaveInfo,
   Envelope)
```

This type apart from the name is in fact the final type we use within Musikell. In the type also we use two synonyms for our two lists of doubles, these are `SamplesGathered` and `SamplesGenerated`.

```
type SynthState = Behavior ( SamplesGathered,
   SamplesGenerated , WaveInfo, Envelope)
```

3.1.6 Modeling Time

Now that we have shown how we decided upon the type `SynthState`, it is time to address another problem that arose when designing the code, when sampling from `samplesGenerated` how would we best represent time. For this we decided that there were two main cases that had to be addressed. The case of sampling for an application in realtime, that is sampling at a rate where human interaction with a wave can be heard instantly. This would require sampling at a faster rate or sampling less values per sample time. The second notion of time we thought of as "file playback", in that all the samples were written to a file and then played back by an external application. This notion of playback brought with it the idea that samples only had to have a regular time step and when values of samples was of little consequence.

One thing we wished for Musikell, is for it at some time in the future to be capable of realtime interaction of a wave, however this was not a major undertaking for this part of the project, as such we could take liberties with our timer. As a result we chose to use the timer that comes bundled with the wxHaskell framework. The reason for this was

two-fold, first it was easy to integrate into the application, thanks to Reactive-banana's focus on working with other graphical frameworks[3]. The second reason was that it worked straight out of the box, which for implementing the rest of the framework was exactly what we needed.

3.1.7 Playback

File Playback

Finally one more design decision that had to be made was how to best store the values of the file. As in the end we use a large series of values it is quite simple to write out the file in many different ways. Continuing with our trend of keeping our framework as simple as possible, we chose to use a pre-existing library for writing sound files, the library was Data.WAVE. The reason for choosing this was that it was designed to write out a list of points easily, and contained a number of helper methods for dealing with some of the more difficult things to do such as defining a `.WAV` header file and the sample rates.

One drawback to using this library however, was that we were very much limited to the `.WAV` file-type. As mentioned these types of file, require a header and a sample for adequate playback, also a restriction that was imposed by the library is that all the data has to be written out at once. This prevents us from streaming values and instead when we gather samples we must append them to an ever increasing list. This of course means that when we come to write out the file, the longer we have been playing a sound the longer the list, and the longer the time it takes to write out to a file.

Realtime Playback

In the later times of this work, we attempted to implement realtime streaming of the sounds as they were created. Unfortunately there are no ready-made libraries to handle such a thing in Haskell. To try and get around this, we attempted to use Sound exchange (SoX) an open source, cross platform, command line application. SoX allows users to playback multiple different types of file, but more importantly with SoX one is able to read in values from its `stdin`, by piping the output from Musikell[1] into SoX we had hoped we would get the sound playing back in realtime, however we were faced with some technical difficulties. We believe that with more time and research in this area we could easily have discovered the source of the problem, however with the limited time we focused on other matters.

[1]To do this, we changed from writing to a file to writing to Musikell's `stout` we also changed the values from being accumulated to streaming

3.2 Implementation

In the previous section we discussed a number of the design decisions that we made, and how we implemented several of the concepts that were mentioned. This section details some more of the technical details that were not mentioned.

3.2.1 Filters and Filtering

A filter in general terms is a component that alters a wave in some fashion, so we could broadly speaking consider a Voltage Controlled Amplifier as a filter. Normally however, we would consider filtering as a function that affects the Harmonic components of a compositional wave, through their attenuation. Filtering can bring new tones to a wave, for example, if we wished for a Saw-wave to have a darker, more muffled tone, we can filter out the higher harmonics of the wave [6].

There are two types of basic filter, high pass filters and low pass filters. High pass filters are filters that will block out lower frequency waves, while keeping higher level ones. Low pass filters as one would expect are the inverse of High pass filters. We can separate the frequencies that are filtered into three separate categories.

- Frequencies within the pass band, these frequencies are the ones that are not affected by the filter.
- Frequencies within the stop band, these frequencies are the ones that have their signal attenuated down to 0, effectively blocking them.
- The transition from the stop band to the pass band can happen gradually, this is area is called the transition band. Values within this band are attenuated based on a interpolation between stop band and the pass band.

We implemented high-pass and low pass filters in Musikell;

```
hPassFilter :: Double -> Double -> Double -> Double
hPassFilter lowCut highCut frequency
      | frequency < lowCut = 0
      | frequency > highCut = 1
      | otherwise = interpolated
                  where interpolated =
                     (frequency-lowCut)/(highCut - lowCut)
```

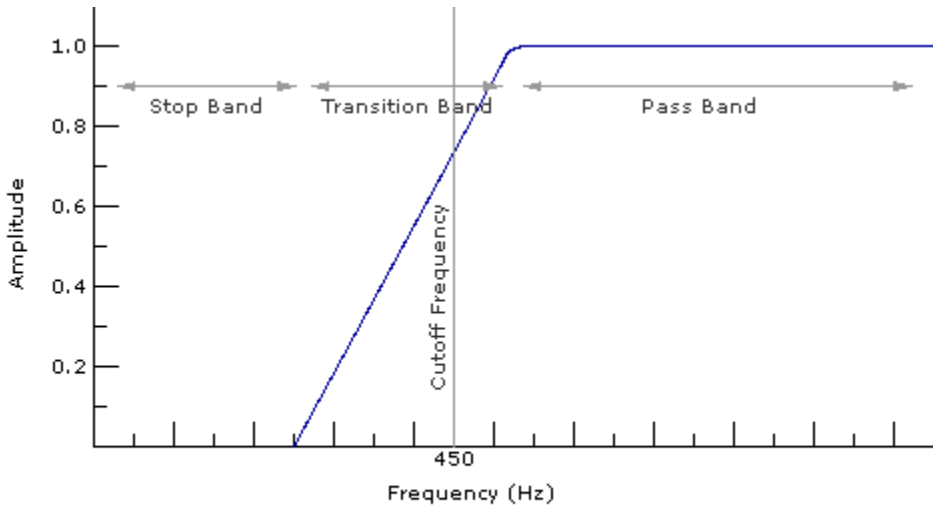

Figure 3.6: A high pass filter, note how the movement from the stop band to the pass band is gradual.

The above function is our high pass filter, it returns a single Double based on the position of the frequency with regards the three bands discussed above. If it is below a cut off frequency we simply return a zero, if it's above it, we return a one. If it's in the transition band we work out using interpolation how far along the band the frequency is. The values returned are then used to attenuate the frequencies during construction.

As low pass filters are the inverse of high pass filters, we are able to define it simply as;

```
lPassFilter :: Double -> Double -> Double -> Double
lPassFilter lowCut highCut frequency
      = 1 - hPassFilter lowCut highCut frequency
```

More complex filters also exist. Two such filters are the band-pass filter and the band reject filter. These filters are created through the combination of a low-pass filter and a high-pass filter. Band-pass filters isolate a group of frequencies and allow them through, blocking all other frequencies. Band-reject filters are the inverse of Band-pass filters and allow all frequencies that are not within range to pass while blocking those within a range. A fourth variable was added to allow for someone to define a transition band for the overall filters. Once again, our band reject filter is just the inverse of the band pass filter.

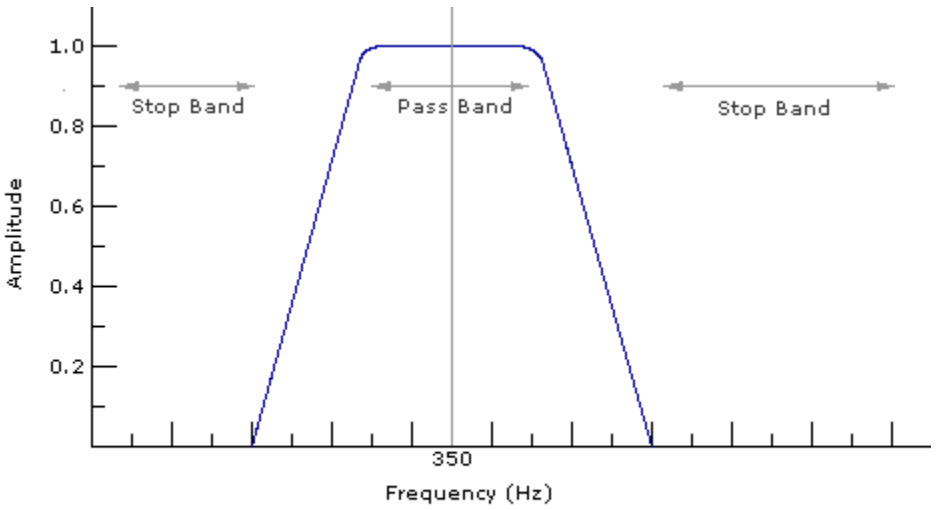

Figure 3.7: A band pass filter, it only allows frequencies within the pass band to sound, all other frequencies are blocked.

As these filters are combinations of the previous two filters. They are easily defined;

```
bandPassFilter :: Double -> Double -> Double -> Double ->
   Double
bandPassFilter lo hi f tran
      = (hPassFilter lo (lo+tran) f) * (lPassFilter (hi-tran)
         hi f)

bandRejectFilter :: Double -> Double -> Double -> Double ->
   Double
bandRejectFilter lo hi f tran
      = 1 - bandPassFilter lo hi f tran
```

Our approach to filtering is different to the standard way. Synthesisers normally will take a composite wave and deconstruct it into it's fundamental frequency and harmonics using a Fast Fourier Transformation. They will then filter these values before re-constructing the new wave. We decided however that a better way to do this for Musikell, was to make a filter part of the waves generation. We could then attenuate any frequencies that were blocked by the filter and generate the wave as normal.

The rationale behind this was that, considering we generated waves through additive synthesis it would be inefficient to decompose the wave again, only to add it back together.

A drawback however is that when a filter is applied we need to recalculate the wave from scratch, and sample from the wave.

3.2.2 Gathering the Samples

Now that we have discussed the other functions making up Musikell, the final thing in the core functions. Is the function `gatherSamples`. This function is executed whenever the timer goes off.

```
gatherSamples :: Int -> SynthState -> SynthState
gatherSamples sample (accum, rest, (WaveInfo s wave cv
   False), eg)
      = (newAccum, newRest, (WaveInfo (s +fromIntegral
         sample) wave cv False),egRest)
      where
            newAccum = accum ++ (force values')
            (values, newRest) = splitAt sample rest
            (egValues, egRest) = splitAt sample eg
            values' = zipWith (*) values egValues

gatherSamples sample (accum, rest, (WaveInfo s wave cv True),
   eg)
      = (newAccum, newRest, (WaveInfo (s+(fromIntegral
         sample)) wave cv False), egRest)
      where
            newAccum = accum ++ (force values')
            (_,newRest')
             = splitAt (round s) (generateWave sample
                (WaveInfo 0 wave cv True))
            (values, newRest) = splitAt sample newRest'
            (egValues, egRest) = splitAt sample eg
            values' = zipWith (*) values egValues
```

As mentioned before depending on whether WaveInfo is carrying a True or False value, this function does two slightly different things. If WaveInfo carries a False, then it means that the currently the sound wave has not been modified and we can continue to sample from the same list.

If WaveInfo carries a True however then we must start sampling from a new wave. Either way the methods for sampling are similar, using samplesPS we split the infinite list into two, which gives us our new samples, and the remainder of the list.

While we take from the list of wave values, we also take from a second list, the envelope generator list, we do this in the same fashion as we do for the wave list. We then multiply these two values, to get the final values for our samples. We then append these new samples to the previously collected samples.

One thing to note, is that we strictly evaluate the samples as we take them from the list, we do this to prevent a time leak which is discussed more in the next chapter.

3.3 Reactive Banana and Musikell

In the previous sections we discussed the main core components that make up Musikell. This section now looks at how we modeled interaction between these components and the outside world.

3.3.1 Events

In Chapter 2, we discussed Events in the context of Functional Reactive Programming in general, but also how they relate specifically to Reactive-banana. To remind others, an event is some occurrence that can happen at a discrete time, be it a button press on GUI, or a mouse press.

With Musikell, we have left it open for future users to decide what different inputs will signal an event in the program. However to best explain the ideas of events and how they act upon the main state, we have included a small runnable application along with this source code. It is important to stress that this is intended as just a proof of concept to demonstrate some of the synth core functionality, and to help explain the key ideas presented in the documentation.

In the example provided Musikell takes it events from the user interface provided. To extract the event from the buttons on the interface, Reactive-banana provides the function `event0`, which can be used as follows;

```
eStartE <- event0 startE command
```

The input to `event0` is simply the widget we wish to register to the event. Now whenever the start button the user interface is pressed an event will be fired. The event

fired is of type `Event ()` which is not very useful to us, as it cannot modify the state. In order to bring in the ability to modify the state we need to replace () with a function instead. To do this with Reactive-banana is again very simple, we use the `(<$)` operator, it's type signature is;

```
(<$) :: Functor f => a -> f b -> f a
```

An example of this in use is;

```
(beginGenEnvelopeE samplePS envelope) <$ eStartE
```

Now whenever the event `eStartE` occurs the () is replaced with the partially applied function `beginGenEnvelope`. We can now take this partially applied function and apply it to the state type using `mapAccum`. To remind ourselves `accumB` allows users to apply functions stored in events to types stored in Behaviors. The Behavior that we wish to apply this function to is of course the SynthState.

```
bSynthState :: Behavior t SynthState
bSynthState =
      accumB ([], generateWave samplesPS rw , rw, (repeat 0))
                 ((beginGenEnvelopeE samplePS envelope) <$
                    eStartE)
```

Now whenever the "Begin Envelope" button is pressed on user interface, an event occurs which updates the SynthState using the function `beginGenEnvelope`. In order to expand this to multiple inputs is trivial at this point. One only needs to register a new event coming from somewhere, e.g another GUI button, or a key stroke, and then attach another one of the functions mentioned earlier to that new event as just discussed. One final step of course is to make sure that the multiples events are all fired as they come in, the way to do this is through `union`.

```
-- Assume that the events have been registered previously
bSynthState :: Behavior t SynthState
bSynthState =
      accumB ([], generateWave samplesPS rw , rw, (repeat 0))
                 ((beginGenEnvelopeE samplePS envelope) <$
```

```
                    eStartE)
                  `union` (endGenEnvelope samplesPS (1000,0) <$
                    eStopE)
                  `union` (gatherSamples samplesPS <$ eAlarm)
```

As we mentioned previously, the functions are all partially applied. The reason for this can be seen by examining the type signature of `accumB` again.

```
accumB :: a -> Event t (a -> a) -> Behavior t a
```

It requires that the functions take and return the same type that is currently stored in the behavior. All the functions with Musikell take the samples per second as a parameter, and some others take more parameters, so it is important to partially apply the functions up to the point where all it requires is the SynthState as input. Once we have ensured this, then we can add in more functions to modify the state easily. This allows for an easily extensible framework.

Chapter 4

Discussion

In this chapter we discuss our experiences with Functional Reactive Programming, with an emphasis on Reactive-banana. We also discuss our thoughts on Musikell as an application, both its strength and weaknesses.

4.1 Functional Reactive Programming

4.1.1 FRP vs FP

Programming using Functional Reactive libraries allows programmers to reason about suitably more complex systems without the need to understand the underlying implementations of how the program deals with data flow. A programmer can reason about these ideas using the abstract ideas of Behaviors and Events if we've subscribed to the "classical" style of FRP, or using Signals and Signal Functions if subscribed to the "Signal" style. By freeing the programmer of the underlying implementation we are able to write shorter, more maintainable code.

However FRP also suffers because of this abstraction, as the concepts that are presented at first can be difficult to understand and often ideas are presented with an assumed prior knowledge, this coupled with the lack of documentation can often leave people new to the subject feeling out of their depth.

This said, due to the fact that we can trace FRP semantics back to the ideas expressed in "Functional Reactive Animation"[10], once a grasp of the high level concepts has been achieved it is quite easy to move between the different libraries, and understand the concepts put forward. Surprisingly, the two different styles of FRP are a boon to someone first learning the concepts, as if they are having difficulty with one style, they may find it easier to understand what is being expressed, under a different light.

4.2 Reactive-banana

In Chapter 2, we gave a summary of Reactive-banana's library. In this section we discuss the appropriateness of Reactive-banana used as a general developmental library and also it's appropriateness when applied to the domain of Sound Synthesis.

4.2.1 Overview

Reactive-banana is a very powerful library that allows users to model many different applications using the concepts put forward by Conal Elliot. As with any library created their are always benefits and drawbacks to using it.

There are numerous benefits to using Reactive-banana, due to its similar semantics to Fran[4], once the idea of FRP is understood properly it becomes a very powerful and simple way to express ideas using only Behaviors and Events, combining this with Reactive-banana's emphasis on Graphical User Interface creation, we are able to not only easily create front ends but also create links between the interfaces and the underlying code using Events. An example of this can be seen in this excerpt from the code `Arithmetic.hs`.

```
-- Defining the layout of the text boxes using wxHaskell
 f       <- frame [text := "Arithmetic"]
 input1  <- entry f []
 input2  <- entry f []
 output  <- staticText f []

 set f [layout := margin 10 $ row 10 $
   [widget input1, label "+", widget input2
   , label "=", minsize (sz 40 20) $ widget output]]

 -- extracting the values from the text boxes and modeling
    them as Behaviors
binput1 <- behaviorText input1 ""
binput2 <- behaviorText input2 ""
```

As we can see we define the text boxes and then simply extract their values so we can reason with them as we would with any previous behavior. A review of the `Counter.hs` code shown in section 2 will show that a similar method can be taken to create events

from buttons in the user interface.

There are drawbacks however to using Reactive-banana. The main major drawback is that unlike other FRP libraries there is no implicit notion of Time in Reactive-banana, this forces many users into creating their own model of time, or relying on other libraries to provide timers for them to use. We believe that Reactive-banana would benefit greatly from having it's own concept of time freeing it from relying on other libraries (as it does in many of the examples given on the wiki page). Another drawback, is that because it is easy to integrate plain Haskell code into Reactive-banana it is easy to mis-use Reactive-banana, by relying on Haskell functions it is possible to introduce State and Time-leaks into programs, two things that Functional Reactive Programming promises to remove, however we feel this is less an issue with the library and more one that falls on the programmer to avoid.

4.2.2 Learning Curve

Reactive-banana compared to other FRP libraries is easily the most well documented of all the libraries. There are numerous examples of small applications, the Hackage entry for it also has a lot of information. However despite all this, there is a difficulty to learning to use Reactive-banana. When first presented with all the ideas, it is easy for someone to feel overwhelmed by the amount of new information. The examples presented while numerous can often confuse as much as aid the programmer as there can be other parts to a program (such as interface creation) before one reaches the section that pertains directly to Reactive-banana. This we felt could lead to in-experienced FRP users to get entangled in the wrong portions of code and miss the point of what Reactive-banana is trying to achieve.

Reactive-banana like other FRP libraries, suffers from the problem that it deals with very abstract, high level concepts, which can make it difficult to understand at first, we are presented with ideas of how to create a network using the Moment monad, without a concrete example of why we need to use this monad and the benefit it provides. Similarly the construction of a program and how to best interact with Behaviors and Events can be lost within the numerous functions presented in the API, as such beginners may very well be tempted to "create the wheel" so to speak. However as stated this is a problem that seems to be inherent with most FRP libraries.

4.2.3 Ease of Use

Despite the learning curve associated with Reactive-banana, once a programmer has come to grips with the flow of a Reactive-banana program, and has become more familiar with the numerous functions provided, they will find it very easy to create simple programs using the concepts. Programs designed around a front-end are very simple to create and programmers can design prototypes quite quickly and simply using the library.

4.2.4 Time-Leaks

Time-leaks are a situation in which a value becomes dependent on previous values for an arbitrary time interval, and these values are not evaluated as produced, thus once a value is required it may take an arbitrary amount of time to calculate this value. The unpredictability of when a program stalls is one of the biggest concerns with regards Time-leaks, as an event may occur sporadically in the lifetime of a program, and that event could force a calculation of multiple values that have been built up in the interval, causing the program to halt.

One of the motivations for expressing FRP (Functional Reactive Programming) as signal functions is to avoid time leaks [16]. Reactive-banana has also designed many of its functions with the explicit concern of avoiding time leaks[2]. However there are cases when a programmer can accidentally introduce time leaks into their programming. As mentioned in the previous chapter, we force strict evaluation of values when sampling from the lazy list, this was done to avoid a time leak in the application when one eventually writes out to a file, as suddenly the program seizes while it tries to calculate the values. Interestingly enough this forced strictness has no negative effect on the run-time performance of the code. It will calculate values at the correct speed, and when writing out to files the time to do so is greatly reduced. It should also be noted that if values were streaming, this strict evaluation during sampling would be redundant as values would be forced to evaluate as they were produced, thus avoiding the time leak.

4.2.5 Appropriate for Application Domain?

Having created a simple sound synthesiser using Reactive-banana, we believe that it was perhaps not the best choice of library for the problem. This is mainly due to the fact that Reactive-banana does not have an implicit time built into it's library. This led to several of the design decisions made, including our choice of how to sample values, because we could not easily integrate a timer into the creation of sine waves as was done with the

synthesiser created in Yampa. We also believe that Reactive-banana although expressive is not as well suited for modeling the ideas of the data flow of a synth as well as Yampa can, by this we mean using Yampa's arrowed framework is a more natural representation of the how sound signals would travel in a synthesizer.

This being said, despite Reactive-banana's shortcomings in this area, it was never designed to be used in this type of scenario, this was not immediately obvious when first choosing the library, and by the time we realized this it was too late to start with a new library. However, we have proved that it is possible to design more complex applications using Reactive-banana, and it gives us hope that in the future we will see a more refined version of Reactive-banana that allows for more general FRP application programming.

4.3 Musikell : An Analysis

Musikell is a very simple framework, and that is both its strength and its weakness. What follows is a discussion on what I find to be Musikell's greatest strengths and biggest shortcomings.

4.3.1 Strengths

Musikell was designed to be a simple and easy to use Framework that can be used to create different sounds, and we believe that we achieved this to a good extent.

One of Musikell's main strengths is how lightweight it is, the codebase is very small, the functions are very straightforward, and though designed with the idea of functional reactive programming in mind, we stepped away from that to some extent and many of the functions do not rely on the necessity of Reactive-banana. Due to this, Musikell is also easily extensible, a person need only add new functions that manipulate the synth state to achieve a new effect on the sound.

Perhaps, amongst the greatest things about Musikell is that it's main functions are completely de-coupled from any reliance on a user interface. Using Reactive-banana or some other FRP library we can create any type of user interface we desire, such as buttons to be pressed on a graphical user interface, or keys being pressed on the keyboard. We need only model these ideas as events that change a behavior in some way and we can have any sort of link up that we desire.

Finally Musikell, is completely system independent. This was one of the main things that we wished to see happen when making the Framework. Some of the other synths, require that one has support for Advanced Linux Sound Architecture, instantly limiting

the platforms that the synths can operate on. Musikell makes no assumption about which platform you work on, only that you have the basic requirements to run the code, in this case Reactive-banana and its dependancies.

4.3.2 Weaknesses

What follows is what we see as the largest shortcomings of Musikell and solutions on how we could perhaps improved upon them if given more time.

The first problem we believe needs to be addressed is the fact that when sampling the infinite list of values we rely on the use of wxHaskell's built in timer. As stated in the strengths section we do not necessarily need to use wxHaskell or any other GUI framework, as Musikell can be independent of any user interface framework, however if one chooses to peruse this route of independence , they will need to devise their own method of sampling values. We believe that given more time, we could have implemented our own variation on a timer, that would have made Musikell more self-reliant, but due to lack of time it was an unnecessary addition to the codebase.

A main goal that we had hoped to achieve but was not possible was that of real time playback of sounds, so that we could hear the change of sound instantly as opposed to listening to it from a file after writing out the samples taken. One of the main reasons for this shortcoming was the lack of Haskell based realtime streaming libraries. I believe this lack of library is due to Haskell's level of abstraction, at this level, it is hard to make assumptions about the underlying architecture and thus hard to implement. One solution that we tried to remedy this problem was the usage Sound eXchange (or SoX) a cross platform command line utility that can allow realtime playback. This seemed promising however experiments in this approach experienced problems. We believe this problem lay in the fact that audio bit rates were not lining up correctly, more experimentation in this area would in our opinion lead to a proper audio real time playback area.

The last major shortcoming of Musikell that I believe needs to be addressed is that lack of concurrency used within the application. Given more time, this would have been one of the major points that would have been addressed, as with concurrency not only comes optimizations to do with writing and generating files, but also we could have addressed the problem that only a single note could be played back a time by Musikell.

Chapter 5

Conclusion

This dissertation investigated the plausibility and practicality of building a simple sound synthesiser framework using the concepts of Functional Reactive Programming, in particular the use of the library Reactive-banana. In chapter one we gave a brief overview of this work as a whole, before examining the origins of Functional Reactive Programming and its libraries in chapter two. In chapter three we outlined the design of Musikell, considering the different decisions and limitations that shaped the framework. Finally in chapter four, we discussed the use of Reactive-banana as an appropriate library for this work, before analysing the success and shortcomings of Musikell.

With regards the aims of this research, we have achieved a moderate level of success with regards a sound synthesis framework. To re-iterate the sentiment in chapter four, Reactive-banana may not be the best suited Functional Reactive library for this application domain, however we have shown that it is possible to create a simple framework based on the concepts provided by the library, and these ideas can be extended into other areas not just sound synthesis, if one so desires.

5.1 Future Work

Although we have built a framework for sound synthesis, there are further endeavours with which one can partake for the improvement of Musikell.

5.1.1 Timer

An intent of Musikell was always to make it as self-contained as possible, currently if we wish to sample data from the list of values (as outlined in chapter 3) we can do so with the aid of a timer provided by wxHaskell. A solution that we would like to see come to

fruition would be to create our own notion of time, and use this new timer to sample values, freeing Musikell from its reliance on other frameworks.

5.1.2 Realtime playback

Realtime playback was not a major concern when considering the creation of the framework. However a future addition to the framework could be a set of functions to achieve this end. This addition would continue the idea of making Musikell as self-sufficient as possible.

5.1.3 Arrowed Musikell

As stated in Chapter two, a similar concept to Musikell had been explored in the paper Switched-On-Yampa, however the design of their system was aimed at the manipulation and playback of midi-files as opposed to sound creation. A possible future for Musikell, would be to re-construct the the ideas presented in this work, using Yampa or another signal based FRP library, to see if better results could be achieved.

Bibliography

[1] Edward Amsden. A survey of functional reactive programming. *Unpublished*, 2011.

[2] Heinrich Apfelmus. Dynamic event switching. `http://apfelmus.nfshost.com/blog/2011/05/15-frp-dynamic-event-switching.html`.

[3] Heinrich Apfelmus. Heinrich apfelmus' blog. `http://apfelmus.nfshost.com/`.

[4] Heinrich Apfelmus. Push-driven implementations and sharing. `http://apfelmus.nfshost.com/blog/2011/04/24-frp-push-driven-sharing.html`.

[5] Heinrich Apfelmus. Unambiguous choice implementation. `https://groups.google.com/forum/#!topic/haskell-cafe/71Tucz2uL0g`.

[6] BeauSievers. Building a synthesiser. `http://beausievers.com/synth/synthbasics/`.

[7] Denis Caromel and Yves Roudier. Reactive programming in eiffel. In *Object-Based Parallel and Distributed Computation*, pages 125–147. Springer, 1996.

[8] Mun Hon Cheong. Functional programming and 3d games. *BEng thesis, University of New South Wales, Sydney, Australia*, 2005.

[9] Antony Courtney, Henrik Nilsson, and John Peterson. The yampa arcade. In *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, pages 7–18. ACM, 2003.

[10] Conal Elliott and Paul Hudak. Functional reactive animation. In *ACM SIGPLAN Notices*, volume 32, pages 263–273. ACM, 1997.

[11] George Giorgidze and Henrik Nilsson. Switched-on yampa. In *Practical Aspects of Declarative Languages*, pages 282–298. Springer, 2008.

[12] P. Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys (CSUR)*, 21(3):359–411, 1989.

[13] Paul Hudak et al. The haskell school of music. *Yale University*, 2008.

[14] Paul Hudak, Tom Makucevich, Syam Gadde, and Bo Whong. Haskore music notation–an algebra of music–. *Journal of Functional Programming*, 6(03):465–484, 1996.

[15] Miran Lipovača. *Learn you a Haskell for great good!* No Starch Press, 2011.

[16] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 51–64. ACM, 2002.

[17] John Peterson, Paul Hudak, and Conal Elliott. Lambda in motion: Controlling robots with haskell. In *Practical Aspects of Declarative Languages*, pages 91–105. Springer, 1998.

[18] John Peterson, Paul Hudak, Alastair Reid, and Greg Hager. Fvision: A declarative language for visual tracking. In *Practical Aspects of Declarative Languages*, pages 304–321. Springer, 2001.

[19] Henning Thielemann. The reactive-balsa package. `http://hackage.scs.stanford.edu/package/reactive-balsa`.