FUNCTIONAL REACTIVE MUSICAL PERFORMERS

A Thesis

Presented to

the Faculty of California Polytechnic State University

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Justin Phillips

December 2010

© 2010

Justin Phillips

ALL RIGHTS RESERVED

COMMITTEE MEMBERSHIP

TITLE: Functional Reactive Musical Performers

AUTHOR: Justin Phillips

DATE SUBMITTED: December 2010

COMMITTEE CHAIR: John Clements, Ph.D.

COMMITTEE MEMBER: Aaron Keen, Ph.D.

COMMITTEE MEMBER: Zoë Wood, Ph.D.

Abstract

Functional Reactive Musical Performers

Justin Phillips

Computers have been assisting in recording, sound synthesis and other fields of music production for quite some time. The actual performance of music continues to be an area in which human players are chosen over computer performers. Musical performance is an area in which personalization is more important than consistency. Human players play with each other, reacting to phrases and ideas created by the players that they are playing with. Computer performers lack the ability to react to the changes in the performance that humans perceive naturally, giving the human players an advantage over the computer performers.

This thesis creates a framework for describing unique musical performers that can play along in realtime with human players. FrTime, a reactive programming language, is used to constantly create new musical phrases. Musical phrases are constructed by unique user programmed performers and by chord changes that the framework provides. The reactive language creates multiple musical phrases for each point in time. A simple module which chooses musical phrases to be performed at the time of performance is created.

Contents

List of Figures	vi
1 Introduction	1
2 Music Theory	4
2.1 Example	7
3 Related Work	9
3.1 Haskore	9
3.2 Music	10
3.3 Functional Reactive Programming	12
4 Implementation	15
4.1 Signal Handler	19
5 Experiment	23
5.1 Performers	23
5.2 Musical Value-Functions	24
5.3 Music Performed	26
6 Results	28
7 Future Work	36
8 Conclusion	39
Bibliography	41

List of Figures

2.1	Twelve Bar Blues Example	8
3.1	Haskore Example	10
3.2	FrTime Example	13

Chapter 1

Introduction

Computer music has been evolving since the 1960's where it began with Max Matthews' MUSIC software. The introduction of MIDI in the 1980's standardized a representation of music for synthesizers and computers. Currently, musicians will perform onstage with a laptop running a descendant of MUSIC named Max after Max Matthews. Max allows users to manipulate the flow of music or sound on a screen just like a producer might connect reverb units and compressors in a studio with patch cords. What Max does not do is provide accompaniment to performers on stage. Loop pedals can replay music that a human has performed, but music composition or creation still lies with human performers. The problem of creating a computer performer has been addressed previously with a system called *Band-out-of-a-Box* [10]. *Band-out-of-a-Box* uses learning to create computer performers which are modeled after performances that are provided to the system through sheet music. Unfortunately *Band-out-of-a-Box* computes offline creating performers after a piece has been performed. The work presented here tries to create performers online, or while the piece is being performed.

Functional Reactive programming is a programming paradigm that extends

functional programming with the notion of signals. Signals are values that vary depending on some parameter and function. Time and seconds are the classic example of a signal. Just as Max uses the flow of music to create and manipulate sound in realtime, signals can be used to manipulate data or functions in realtime. FrAN, a animation library for Haskell, is the first implementation of a functional reactive programming paradigm. FrAN is able to express complicated graphical animations with mathematical functions which define ‘x’ and ‘y’ coordinates of an image or graphic on screen. FrAN has since evolved into other libraries (see Reactive and FieldTrip) and Functional Reactive Programming, or FRP, has since evolved into its own programming paradigm, separate from the graphics domain. Different implementations of FRP use different methods to propagate changing signal values through the system. Most recently a push-pull hybrid method was developed in Haskell [4].

The system that we develop uses the language FrTime (pronounced “Father Time”), a Functional Reactive implementation built on top of the language Racket. A parser transforms MIDI signals into Haskore which is a way to represent music that was developed for Haskell [7]. The music is then analysed to find the key signature and chord changes. A user created performer creates music to be played over upcoming chord changes that the system predicts. Finally the system performs a musical phrase using a evaluation function to choose one out of the many possible phrases to perform that are created anytime a new note is played.

A short introduction to music theory is provided followed by related works, both in the field of computers and music, and in the functional reactive domain. Following that is a description of the implementation and problems that arose and how they were solved. The fifth section describes an experiment in program-

ming different performers followed by a section describing the results of these experiments. Future work is discussed and followed by a summary of the work.

Chapter 2

Music Theory

Music can be described as a series of notes happening at discrete times. More specifically, a piece of music consists of four characteristics: time signature, tempo, key signature and form. This representation neglects other characteristics of music, like timbre and dynamics, that describe the performance of the notes. We are only concerned with the notes, their timing, and the order in which the notes occur as events.

Notes are the union of three different characteristics: pitch, duration, and location in a score. A pitch is a pitch class like ‘C’ or ‘F#’ paired with an octave which is just an integer representing the number of octave. Middle ‘C’ for instance is the pitch class ‘C’ paired with the number 4. The octave number is relative to an 88 key piano where the low ‘A’ begins the first octave. The duration is simply how long the note lasts, represented in time through the use of time signature and tempo. The location in the score is when the note happens giving the note a starting time in relation to other note events in the score. A measure of music is a grouping of notes for a certain amount of time. A measure of music can have any number of beats other than one or zero.

The time signature describes how many beats are in a measure and what kind of beat they are. For example, $3/4$ and $4/4$ are common time signatures. A time signature of $3/4$ means that there are three quarter notes in each measure. A time signature of $4/4$ then would be the same, except that there would be four quarter notes in each measure. The rate at which the beats happen in time is called the tempo and is measured in beats per minute. A song in $4/4$ can have a tempo of 120 or 60 or any other number. In the case where the tempo was 60 that would mean every quarter note would be played every second because 60 beats per minute converts to one beat per second.

It should be noted here that computers and humans differ distinctly on their respective interpretations of music being played on the beat and at tempo. This will be discussed later in [section 4](#).

Music is also made up of different frequencies called notes or pitches. Humans understand pitch in terms of the distances between them. We do notice the difference in frequency or pitch when provided with a reference. We can distinguish between a difference that is large and a difference that is small.

In modern Western music all of the pitches used in a song are members of some scale. A piece of music is said to be in a specific key and has a key signature. A key signature acts as convenient way of noting which pitches will be most commonly used in the score of music. Each key signature also has a corresponding scale. Every scale has a pitch class and a mode. The only modes we are concerned with are the major and minor modes and the difference between major and minor scales are the pitches that are considered members of the scale.

‘C’ major is the key signature that encapsulates all of the white keys on the piano. The members of this set, called a scale, are ‘C’, ‘D’, ‘E’, ‘F’, ‘G’, ‘A’,

and ‘B’. In addition to just the pitches, specific chords can be part of a scale. Continuing with our example of ‘C’ major, a ‘C’ major chord has the notes ‘C’, ‘E’ and ‘G’ all played at the same time. This chord also is a member of the key signature ‘C’ major and exists in the scale of ‘C’ major.

When humans infer what scale is being played or what the key signature is we do it by hearing multiple pitches, either in chords or in musical phrases. If I hear the chords ‘C’ major, ‘D’ major and ‘G’ major I infer that the piece of music is in ‘G’ major. If I hear the notes ‘D’, ‘E’, ‘F#’, ‘G’, ‘A’, ‘B’, and ‘C’, I infer that the piece of music is in ‘G’ major. The chords and pitches may be played in any order.

Chords can be put together to make a chord progression. Chord progressions are simply patterns of repeating chords. ‘C’, ‘G’, ‘F’, ‘G’, would be a ‘C’ major chord, followed by a ‘G’ major chord, an ‘F’ major chord and finally another ‘G’ major chord. Progressions are a smaller part of what is called the form. Some progressions have what are known as cadences. A cadence is a specific set of chords at the end of a musical phrase that drives the direction of a piece. Three cadences that are used frequently are *authentic* cadences, *half* cadences and *deceptive* cadences. An *authentic* cadence is a chord progression that moves from the chord based on the fifth note of the key signature to the root or first note of the key signature before a strong beat. A short example classified as an *authentic* cadence would be if a piece were in ‘C’ major and in the third measure a ‘G’ major chord (the chord based on the fifth of the key ‘C’ major) were followed in the fourth measure by a ‘C’ major chord. A *half* cadence has the fifth chord played before a strong beat. So the ‘G’ major chord in the previous example would occur in the fourth measure, not the third. Any chord can come before the chord based on the fifth. The *deceptive* cadence is just like the *authentic* cadence

except instead of moving to the root chord after having played the fifth chord the piece moves to any chord but the root or fifth chord.

Most pieces of music do not strictly adhere to a key signature, using “borrowed” pitches from other key signatures to add color and character to the music. This, along with how the computer interprets pitch is discussed in section 4.

2.1 Example

Taking figure 2.1 as an example, we can see a simple twelve bar blues form. Twelve bar blues is the classic chord progression that takes place over twelve measures of music. In this case the piece is in ‘C’ major. The twelve bar blues pattern is just the movement through the first, fourth, and fifth chord before coming back to the first or root chord. This movement can be in any key signature or scale. The fourth chord always occurs in the fifth and sixth measure. The fifth chord always occurs in the ninth measure followed by the fourth chord. The rest of the measures are the root chord except for the last measure which can be the fifth chord again for a turnaround. In this example though the last measure is the root chord again.

A human player can often join in playing with performers just by listening to the music and discovering the chord progression, time signature and key signature by ear. Computer performers have many ways of discovering key signature, chord progression, and time signature. The way that music is described in this system is through a time-signature, key-signature, tempo and chord progression. With these data the computer performer can make a prediction about what chord will be next and perform some music just as a human player might listen before joining in playing.

Figure 2.1: Twelve Bar Blues Example

Chapter 3

Related Work

The following section is organized as follows : a short description of Haskore, related works that deal with music, and finally the work done with Functional Reactive languages.

3.1 Haskore

Haskore is a library written for Haskell used to describe, transcribe, and manipulate music. It uses a simple syntax to describe music and music related events. Haskore provides access to other library functions, in addition to providing a syntax for representing music.

Figure [3.1](#) depicts a simple example of some music in Haskore. The first piece of music is a ‘C’ Major scale where every note is performed for a duration of a quarter note. The second piece of music represents a ‘C’ major chord followed by a ‘G’ major chord with a half note length rest. When music is played ‘in parallel’, meaning it is chorded or there is some amount of time where the two notes are

```
(:+: (note (C 2) 1/4) (note (D 2) 1/4) (note (E 2) 1/4) (note (F 2) 1/4)
      (note (G 2) 1/4) (note (A 2) 1/4) (note (B 2) 1/4) (note (C 3) 1/4))
```

```
(:+: (:=: (note (C 2) 1/4) (note (E 2) 1/4) (note (G 2) 1/4))
      (rest1/2)
      (:=: (note (G 2) 1/4) (note (B 2) 1/4) (note (D 3) 1/4)))
```

Figure 3.1: Haskore Example

sounding simultaneously, the ‘:=:’ symbol is used. Other music, like the scale example, is played ‘in sequence’ and the ‘+:’ symbol is used. The notes have a number value associated directly with the pitch, 2’s and 3’s in the example. This just describes which octave the pitch should be played in. The other number is the duration of the note. The ‘+:’ and ‘:=:’ symbols can be used together, as seen in the chord example, to describe any kind of music. It should be noted that at the time of writing this the work in Haskore seems to be continuing with work in Euterpea here <http://code.haskell.org/Euterpea/>

3.2 Music

The following section details related works in computer music in chronological order.

Computer music problems have been classified as problems with transcription, pitch determination and segmentation, time and frequency analysis, and

most interestingly, problems in discovering a proper musical grammar [5]. If a complete musical grammars were to be defined generation of new music could be programmed. All of these problems have been investigated seriously but not have been effectively solved.

The problem of recognizing melodies and repeating motifs is studied as a string matching problem [3]. A subset of all string matching problems is unique to music and a few other related domains and have been classified. The most relevant string matching problem is the problem classified as “Overlapping Repetition Identification.” The problem consists of locating repetitions in the musical score or string. The solutions that are referenced account for repetitions in different voices, which compute in non linear time, and a more efficient solution of using ‘tiles’ to discover repeating segments. A different approach to the same problem has been attempted with the use of databases [8]. This solution describes a method of finding every repeating pattern in a piece of music. With this method every repeating pattern in a piece of music with 1000 notes can be found in 10 seconds [8].

The problem of creating an improvising computer performer is studied in a program called *Band-out-of-a-Box* [10]. This is a system designed to trade solos between a human player and the computer performer. This system creates a learning model that can be trained on existing jazz improvisations. A player performs improvisations which are then represented by variable-length trees. These trees record every note in every measure and then in the context of the measure describe a *Pitch Class*, an *Intervallic Motion*, and a *Melodic Direction*. The *Pitch Class* is just a tally of which notes were played and how many times the notes occur. *Intervallic Motion* records the intervals, or distances between the pitches in each measure. *Melodic Direction* has a rather complex way of recording descent

and ascent of the pitches. The *Melodic Direction* records the number of times successive pitches descend, remain the same, or ascend, three note sequences descend, remain around the same note or ascend, and finally records the same data with four note sequences.

The work on *Band-out-of-a-Box* also includes an experiment of using the learning framework with the variable-length trees and the three characteristics noted above. The interesting conclusion drawn from the experiment is that these three characteristics, *Pitch Class*, *Intervallic Motion*, and *Melodic Direction* are useful at describing the activity and motion of a performer’s improvisation. Additional processing can be done over the data to discover what scales are used in what measures, how frequently those scales are used with certain rhythms and syncopations, and other interesting relations that can help create interesting performer. *Band-out-of-a-Box* is able to create a very complex improvisation by learning all of these properties.

Finally, another solution to discover and match patterns uses a “sliding window” [6]. The solution described is to have a sliding window of cut pieces. A ‘join’ function is also described which can concatenate patterns that are overlapping resulting from the to the cutting process and the sliding window.

3.3 Functional Reactive Programming

Functional Reactive Programming, or FRP, is a programming paradigm that incorporates reactive elements in a functional language. FRP incorporates signals which are time varying values. A simple example is shown in figure 3.2. The first expression creates a new signal which flip flops between the values of one and zero. The next expression is simply a division function that prevents divide by

```

#lang frtime
(define flipper (modulo seconds 2))
(define (divider x y)
  (if (equal? y 0)
      "divide by zero"
      (/ x y)))

```

Figure 3.2: FrTime Example

zero errors. When flipper is passed as the second argument to divide it creates a new time varying signal. Depending on the value of flipper the divider expression will evaluate to either “divide by zero” or the argument it is passed.

The motivation for using a reactive language is primarily driven by many rapidly changing values and events. In our case the values that change are the inputs (notes performed by the human) and outputs (predicted chord progression) of the non reactive system. A signal is created that creates a new musical phrase to perform every time a note is played. When it comes time to perform the musical phrase a different signal chooses between the possible phrases and performs the phrase it chose.

The languages used for this system are FrTime and Racket. FrTime is built with a completely push-based evaluation model [2]. This choice was made so that the REPL would still work as a Scheme/Racket user would expect it to.

A semantic has been developed to handle indeterminate behavior in parallel programs or programs that are being run on multiple processors [1]. This semantic can handle values yet to be processed, or even non terminating computation by using speculative computation.

Realtime results are necessary for reactive programming which has led to a

formalized semantic which guarantees realtime results [12][11]. This semantic was developed from a desire to use a functional reactive language in robotics applications. The work described in this paper does not expand upon this realtime semantic, but relies heavily on its basis and guarantees. The FrTime implementation incorporates these ideas [2]. The performer must react in realtime to events that are triggered just like a robot must react to an external stimulus in realtime. Luckily, reacting to an external stimulus is the same for a robot and a computer performer. The framework described in this paper would not exist without these previous work’s guarantee of real time evaluation. Also, there is a comparison of real time MIDI performance across different commercially available operating systems not using any reactive languages, but instead trying to compare the performance of each operating system independent of other considerations [9].

Whereas FrTime is developed with a push-based evaluation model the reactive language in Haskell has recently been implemented with a hybridized push-pull method [4]. Signals have to be decomposed into two classes to effectively implement a reactive language with a push-pull hybrid evaluation strategy. The first class, called reactions is discrete. These are events like mouse clicks, key depressions on the keyboard or time-related events that happen irregularly or infrequently. The second class is called time functions and is continuous. These are events that occur repeatedly causing recomputation of a dependency over and over again. The push semantic is used for the data-driven reactive class. The pull semantic is used for the demand-driven time function class.

The idea of a ‘futures’ is also introduced as a time-value pair. Futures are values that are not known yet but can be considered and preemptively computed [4]. A list of futures is called a progress report and can be processed to remove some potential futures, shortening the list.

Chapter 4

Implementation

The main advantages in using a reactive language for the implementation come at the highest level of the implementation. The system uses what essentially reduces to a small set of modules, a parser and an analyser, which are written in Racket. These two modules are used and process new data anytime a new event is received.

The system requires a MIDI instrument connected to the player's computer. On Mac OS X the CoreMIDI library is used to retrieve the MIDI data that the instrument outputs. On Linux, the player must also install Jackd (jackaudio.org) and manually connect the MIDI instrument's out port, through USB or some other interface, to DrRacket's in port. At this point the MIDI data is parsed and concatenated to any MIDI data that has already been seen by the system. The parser outputs a correctly formatted piece of Haskore which is built up as the messages are received.

We chose to use MIDI to avert the complex and unfinished work regarding pitch and time signature. MIDI provides the exact note number, so there is no

need to write signal processing software. The player must input a time signature and a tempo. These are the only data that the system cannot discover from the music itself. Once those data are given though, the system can construct the music with all of the note values correctly described. Analysing the rhythm provides slight problems because of the exactness with which MIDI data can describe time. MIDI provides ‘on’ and ‘off’ signals for each note. So by depressing middle ‘C’ on a MIDI keyboard, the MIDI ‘on’ message is sent. When the key is released then the MIDI ‘off’ message is sent. The duration between the ‘on’ and ‘off’ message is the duration of the note. For a performer playing in the time signature 4/4 with a tempo of 60bpm a quarter note should last for one second. Humans playing with these restrictions will most likely perform notes of different durations, but durations that approximate one second. Since MIDI is so exact it is possible to interpret two quarter notes played in succession as two notes with a rest in between. To account for this human variance a method called quantization is used. Quantization in this system basically allows a range of values to be described as one specific value. So a quarter note in 4/4 at 60bpm can last anywhere from .9 to 1.1 seconds and still be a quarter note.

With the MIDI events parsed and processed into a Haskore structure the musical analysis takes place on the Haskore data to create a musical piece which is a structure that holds possible key signatures, a list of chords (the progression) and a time signature. As the player performs more of the piece the list of chords grows and the list of possible key signatures shrinks. The list of chords will grow indefinitely in the current implementation, but due to the repeating nature of music there is potential to use another piece of data in the piece to describe the form, apart from the chord changes. The key signature is strictly defined as having only notes that are harmonic to all the major and minor keys. That is,

if a song in ‘C’ major is being performed and an ‘F#’ is played as a chromatic passing tone, ‘C’ major would be excluded incorrectly from the list of possible key signatures.

Some color notes will remove a key signature from the list of possible key signatures while others won’t. Since some color tones are frequently used they are considered part of the key signature. Specifically, in a major key the minor third and minor seventh are allowed to occur. These two tones are allowed because of their frequent inclusion in jazz music. In a minor key the major sixth and major seventh are allowed to occur. These tones are allowed due to their inclusion in the melodic minor and harmonic minor scale.

Musical form can be used by humans to predict the next chord that will be played. A human performer hears that other performers are playing the chords ‘C’ ‘d’ ‘F’ ‘G’ in succession and assumes that the song cycles through these chords the whole time. A human is able to do this because of pattern recognition. Computers are particularly good at matching patterns, but the patterns being matched have to be provided.

With the musical piece in hand, the system tries to create a musical fragment for performance. When the performance class is given a musical piece, it tries to predict, from the chord list that it has, which chords will be played by the player next. If it is able to predict some future chords then it writes a musical fragment that would adhere to the upcoming chord progression and performs it. So if a ‘C’ major chord followed by a ‘G’ major chord had previously been performed, then the next time the performer reads that a ‘C’ major chord has just been played it will write music assuming that a ‘G’ major chord is next.

Our implementation presents a problem. Every time that the system receives

an event or a note, the entire piece of music, or every note that has been performed so far has to be parsed over to look for patterns. Instead of worrying about capturing every pattern that exists in the piece of music, the system keeps track of patterns that it has seen and when it sees a new pattern it makes a note of how many measures of music it has seen so that the next time it iterates through the music it knows to start looking past that specified measure. The framework then tries to match the current chord progression against its list of discovered patterns to predict what chords will be played next. The other, non linear methods described are too slow to be useful in a realtime setting. The sliding tile is fast enough, but slows down significantly as the size of the piece increases. This is also true with the database method which used matrices and took 10 seconds to match every pattern for 1000 notes. Since realtime execution is the priority here the sliding window method was simplified to account for just the recently seen measures. Another simplification and optimization is used to enhance the second pass of the music. The first pass finds possible new chord progressions that are part of the musical form and adds them to a list of saved chord progressions. The second pass looks at the last couple of measures in the music and tries to match a discovered pattern to one that has been saved. This too, can slow down the realtime performance if enough patterns are saved. To get around this problem, only the ten most recent patterns are saved.

The whole system works together with FrTime’s event handling infrastructure. A synchable Racket event is made and setup through Racket’s C API . This Racket event depends on MIDI messages. Anytime a MIDI message is sent from the player’s instrument this Racket event is synced. The synchronization of this event sends a FrTime event to a FrTime event receiver. This FrTime event in turn is used perform the music through the use of the ‘map-e’ function. The

‘map-e’ function maps a function that performs music through the speakers to every event it receives.

A different problem arises at this point in the implementation. Since the FrTime event receiver will trigger the execution of the performer with every MIDI message it receives, the system will create too much music. If the performer correctly predicts the upcoming chord progression for the next three chords it will also predict the correct chord progression when there are only two upcoming chords to predict and also only one. In this way the performer will perform the same measure of music as many as three times (the last measure to predict). The other case that is of concern is when the performer incorrectly predicts the upcoming chord changes. In this case, the performer could be playing music that does not adhere to the musical structure and will be distracting for the player and any listeners. Both of these issues reduce to the same problem. The system has many values to choose from at any point in the future and it must decide which value will be the ‘best’ at or before that time.

4.1 Signal Handler

A signal handling module was developed to accommodate the problem of many values to choose from at each distinct point in time. The problem can most simply be described as having many values for each distinct point in time, but only needing one.

This signal handler requires a clock, an evaluation function that takes any number of arguments, a signal reference, and finally a base value. The clock is used to know when the value should change. The signal reference is shared between this signal handling interface and anyone interested in the value that is

produced. This interface also provides a function ‘add-to-signal’ which allows the programmer to add values to the signal at discrete points in the future. These future time values correspond to future time values of the clock that were passed as a reference to the signal handler at instantiation. Finally, the signal is set to the base value whenever the clock ticks and no values have been added to the signal for that instant in time. This prevents the system from performing the same musical fragment repeatedly until a new one arrives.

In a simple example the value-function would be the mathematical function ‘max.’ This function always returns the largest of the values that were passed to it. So if the user of the signal handler called ‘add-to-signal’ with the value 10 at time 1, and the value 3 at time 1, and the value 2 at time 1, then when the clock ticked to 1 the value of the signal would be 10 and only 10 because that is what max would return.

For our use, the discrete values which need to be passed into the ‘add-to-signal’ function should be small musical fragments. The value-function should then choose between any possible musical fragments in the list of added values to choose the most musically appealing value. With this tool the signal will only ever have one musical value at each discrete point in time. That value will then be performed and heard as the output of the system.

This signal handler adds a small amount of pull based or demand-driven evaluation. The semantics of the underlying FrTime language have not changed, and as the FrTime language is built on an entirely push-based or data-driven model, the benefits are slight. The signal handler can be considered pull based because the signal’s possible values are updated by any number of threads or streams of processing and the value for the signal is computed only at the points in time where it is called for. That is, the signal does not waste computation

pruning possible results for any distinct future value. Only when the clock has ticked to a new value is the function, whatever it may be, used on the possible values for the signal at that point in time. If the signal handler was not put in place then the values of the function would be computed at every point where a new value was added, adding overhead that is not necessary. This is similar to the work done in the hybrid push-pull but implemented over a very strict subset of what they accomplished. This signal handler could not handle the case where the list of possible values were non-terminating unless the function provided to compute over the list were able to handle that case. This signal handler could perform any function over the data.

Traditionally, demand-driven semantics are used for evaluating continuous cases and data-driven semantics are used for evaluating discrete cases [4]. The way that the demand-driven model is used in this system is antithetical to that idea. The real gains in efficiency from changing to a hybridized push-pull model are compared to a completely pull-based model. Theoretically, the efficiency of a push only model should be equal, if not a little more efficient, compared to a hybridized push-pull model but they have not been explicitly compared.

This signal handler is weak as a demand-driven or pull based model because it lacks continuity. While it does explicitly pull and update a value it does so only when there are values to be processed. The base value is used whenever there are no values to process because the queue has been left empty for distinct discrete point in time. This ends up being a strength in this application because the framework sometimes has no idea what notes to perform. With the base value set as a rest it is able to be quiet while it processes. If it were continuously evaluating new values then there would be overhead reintroducing the base value at any time the framework was confused.

The signal handler also handles issues that were presented in the parallel programs work where multiple threads or processors handle the same data. Any interested thread can add values to the queue or change the value-function. When the time comes for evaluating the signal's value only one result will propagate through. So if multiple threads or processors were using this signal handler's signal as a base signal then the propagation up the dependency tree would be equivalent across all the threads.

Chapter 5

Experiment

Finding which implementation is most appealing to the user of this system requires multiple executions with different functions controlling the output of the musical signal and different logic in the performer itself. A small experiment was performed with the system in hopes of meeting the author's definition of a musically pleasing performer.

5.1 Performers

Two different performers are used with each of the music value-functions described below. The first one creates music depending on the chord changes that it was provided explicitly. This first performer looks for cadential patterns and writes music specific for those patterns. If no cadence is found it just plays the root of each change it sees for a whole measure. It does this for each note in the list of upcoming changes. If a cadence is found then depending on if it is an *authentic* cadence, a *half* cadence, or a *deceptive* cadence, it makes random music of random durations using the chord changes it assumes are to be performed up

until the cadence, then writes specific music for the cadence taking advantage of the cadence. For example, if it is a *half* cadence it ends the phrase either on the third or fifth of the last chord, which would be the chord based on the fifth of the key signature.

The second performer makes melodies starting from the root note of the first chord that is predicted. It relies more heavily on the key signature than the first performer discussed. It stays in key or keys that have been provided by the analysis. This means that some of the notes will probably be out of the actual key signature. The way it constructs melodies is to randomly ascend and descend the the notes that are harmonic to the possible scales. All notes are given a duration of one eighth note. It performs like this for the duration of the provided upcoming changes. So if it were given a list of three chords it would make three measures worth of music in this fashion, but not necessarily adhere to the chord changes that were provided.

5.2 Musical Value-Functions

Three different value-functions were written for the experiment. Each function was used with each performer.

The first function simply returns the value most recently added to the signal assuming that the performer will write better music when it is predicting chords that are temporally closer to what has actually been performed by the player.

The next functions were inspired by the work done with *Band-out-of-a-Box*. This work closely mirrors the use of intervals and melodic direction although not as completely as was done previously [10].

The second function looks for the most rhythmically similar passages. As musical fragments are performed the duration of each note is tallied, keeping track of which rhythmic values have been most common. If the first musical fragment consists of two quarter notes then that is added to the state. The function chooses passages that have notes over those that do not so that even if the rhythm is more consistent in a passage of nothing but rests, the passage that has actual music to play will be chosen. The next metric for the function is a simple addition of the number of rhythm values that have been seen to the ones to be chosen. So with the previous example, we had two quarter notes. Imagine two passages are being compared in this function. One has a quarter note and the other does not. The passage with the quarter note will be chosen and the state of rhythms that have been seen will reflect the two previously viewed quarter notes, the new quarter note about to be played, and whatever other rhythms the passage about to be performed consists of.

This function uses rhythm instead of pitches or intervals, in the way that *Band-out-of-a-Box* studied. By trying to copy the rhythms it has ‘heard’ it tries to mimic the human performer. A complimentary function could be written that performs rhythms that the human performer is not performing (e.g. triplets if the human plays quarter notes etcetera.)

The third function uses some simple knowledge about melody to choose which musical passage should be returned. Most melodies have a sort of rise or a fall in pitch that takes place over a few measures, giving the melody a contour. This function picks passages that have notes over those that do not for the same reasons as the second function. Initially, the function chooses passages that descend with equal probability as it chooses those that ascend. Once a direction has been chosen though, it tries to continue that direction for three signal values.

After those three signal values the probability to go up or down again is reset to zero so that the melody can rise and fall and does not simply careen off in one direction. This function chooses between different passages that head in the same direction by using a distance metric. Melodies often times jump from one note to a note twelve tones above or below it. Other times, melodies will meander around a central note with small hops and scalar motions. This function chooses between passages that have the correct direction by trying to keep the melodic distance travelled small. When looking for a passage that descends, a passage that descends a distance of ten will always win over a passage that ascends a distance of one. Deciding between two passages that descend, one a distance of ten, the other a distance of three, the less distant descent will be chosen. Distances are defined as the most extreme distance in the passage.

While *Band-out-of-a-Box* used all of the data to learn and train an improviser to act like the one in the piece, this function uses only the data it has ‘heard’ played by itself to choose the next passage to play.

5.3 Music Performed

For simplicity the music that was played for the experiments was all the same and easy to replicate. The human experimenter played in 4/4 at a tempo of 120 beats per minute. Two different phrases were played to determine the different qualities from the different computer performers and music value-functions.

The first phrase that was played was simply a repetition of ‘C’ major and ‘G’ major. After the computer had been performing for twenty seconds or so, the human player began playing ‘D’ major and ‘A’ major chords at the same rate. After the computer had sufficiently adjusted to the new key signature and chords

the human jumped back to ‘C’ major and ‘G’ major.

The other phrase that was performed was more complicated. A simple cadential pattern was developed and cycled through. The chords ‘C’ major, ‘d’ minor, ‘F’ major, and then ‘G’ major were played and then were followed by the chords ‘C’ major, ‘F’ major, ‘G’ major, and finally ‘C’ major again. This is a simple pattern that includes both a *half* cadence and an *authentic* cadence. Each chord was held for two beats or one second.

Chapter 6

Results

This section describes how the music created by the different performers and music value-functions was received by a critical ear. First the simple chord progression of ‘C’ major to ‘G’ major that included a change in key signature to ‘D’ major is investigated. Then a more complicated cadential chord progression is critiqued. The section is organized primarily by the music value-functions. Within the discussion of each function each chord progression is discussed with each performer.

The first chord progression was the simple repetition of ‘C’ major and ‘G’ major which then switched to ‘D’ major and ‘A’ major and finally returned to ‘C’ major and ‘G’ major. The first music value-function, the function that returned the most recently created music was tested with both the cadential performer and the scalar performer. The cadential performer sounded best with this function and the third function with these chord changes. The phrases that it constructed were rhythmically complex, due to the randomness, and disjoint, including a fair amount of rest time. The scalar performer performed modestly playing meandering ascending and descending patterns. With the more complex musical phrase

the music created was aesthetically unbearable. It had no direction and lacked melody. With both of the performers this more complex phrase did not produce any music worth listening to. With the more simple musical phrase this function was not offensive, but equally was not particularly interesting.

The second function, the function based on previous rhythmic values was able to give the cadential performer some much needed rhythmic structure. The melodies created were still abstract at best, having no real direction or personality but by restricting the rhythm just slightly as this function does, the music created sounded less random and did not make large leaps in rhythmic duration as frequently as the previous experiment's phrases did. The scalar performer had no effect with this function as all of the notes are given a duration of an eighth note. This performer did not sound much different here than it did in the previous function. With the more complex musical phrase this function actually produced better results than the previous function. The structure that the function gives by choosing phrases based on rhythmic similarity helps to keep music intelligible. The first performer was able to abstract aspects of performance better than the second performer. The second performer sounded too chromatic and almost atonal due to the freedom of note choice. The second performer could be improved in this case by changing what is defined as notes that are in the key signature. If the key signatures were more limited than this, the second performer would have less keys to choose for this phrase. Since the key signature is loosely discovered over a longer period of time and includes so many color notes the possible key signatures during the performance are still too many and lead to particularly unpleasing chromatic passages.

The third function, the function which chooses phrases for their general melodic direction had the most interesting effect on the scalar performer. The

cadential performer performed poorly with this function because phrases that jumped large intervals were played more frequently, a choice made by the function's desire to continue passages in a general direction. This jumping of large intervals was not pleasant or very creative. The scalar performer on the other hand, which creates randomly ascending and descending patterns was actually given some character with this function. Normally this performer just meanders in either direction. This function gave these meandering phrases some overall direction. The passages that previously meandered now had direction. This was the most musically appealing of all of the different combinations. With the more complex phrase once again, the cadential performer produced better music. The passages that were created had better melodic direction and were less confusing than the first function with the cadential performer. The scalar performer was confused still due to the many possible key signatures. This cadential performer suffered from the same limitation as described in the second performer of having too many possible key signatures. An improvement would be to add some randomness to the rhythm that is chosen for the duration of the notes so that the music does not sound so metronomic.

The first function provided the most consistent results for the simple chord progression. Since this function always chooses the most recent piece of music the phrases are shorter, usually one measure. This function is probably not the best or worst choice for any particular performer that a user could create. The music that it chooses tends to be very consistent though most likely due to the performer's logic for the final chord in a progression being limited by the cadential performer and just the static nature of the music created by the scalar performer. With the more complex chord progression this function performed poorly.

The second function is a fairly good choice as it provides structure to the

music. This is also its greatest weakness though. The music that is chosen can sound annoying and too repetitive. Since it does not use the notes that are chosen, just the duration of the notes to decide which phrase to play, the melody can jump from one place to another. This did produce fairly consistent results with the more complex phrase of music though. The cadential performer in particular produced interesting music that was easier to listen to than the first function. Obviously this function has no real effect on the scalar performer as every note in the passages that are created have the same eighth note duration.

The third function does have an effect on the scalar performer. The music can become boring and annoying though with this performer because the movement tends to be centered around one or two notes. This could be improved by adding a jump or two which would add some much needed randomness and surprise to this performer. This function worked well with the cadential performer as well. This function's weakness seems to be centered around its concern for rhythm. The music choices can leave confusing pauses in the performance due to the randomness that the cadential performer uses in its duration choice. If the cadential performer were less random with its rhythm choice, or this function were more concerned with the note duration the results would probably be the best. Just as the last function did not account for pitch choice this function does not account for note duration. The composition of both of these functions is worth investigating.

With the simple passage of music the best combinations were the scalar performer with the third function and the cadential performer with either the first or second function. When more complex music was used the scalar performer produced unpleasant music. The cadential performer was best with either the second or third function. The cadential performer could be improved by using less ran-

domness in its rhythm choice. If the rhythms that were used were more consistent than the performer would sound less confusing and disconnected. Alternatively, the scalar performer should be more rhythmically diverse. The second and third function could also be combined to make a new function.

Probably the most useful addition to the music value-functions would be some sort of kill switch. Since the performer continued to play phrases in the incorrect key signature after the piece had modulated, the music that it performed sounded particularly dissonant. This was particularly noticeable when the human performer switched from playing ‘C’ major and ‘G’ major to ‘D’ major and ‘A’ major. Since the prediction the framework makes can extend indefinitely, the performer plays music based on ‘C’ major and ‘G’ major chords for close to five seconds before reacting and writing new music with the new ‘D’ major ‘A’ major pattern. The framework predicts correctly that the player will continue to play ‘C’ major and ‘G’ major. It is the spontaneity of music which causes the incorrect performance by the computer. The correctly predicted music is no longer relevant because the form of the piece has grown or moved on.

The music value-functions have no perception of what chord is currently being played and only choose music based on the metric of general musicality over all key signatures and chord changes. A secondary function could be included into each music value-function that would run before the functions experimented with ran. This secondary function would take the list of possible music to perform and the chord changes that the music is created with and compare it to the current state of the analysed piece of music. If the upcoming changes are different from the chords that were currently being performed the music that was made with those changes will be excluded from the second pass, the music value-functions that decide which to play based purely on musical quality. This is an extension

of the idea of progress reports that was mentioned in the push-pull hybrid work. An infrequent polling could take place over futures, the time-value pairs, and remove some from the progress report.

The patterns that are discovered may be too simplistically defined. Since some progressions do not change chords every measure the framework can consider not moving from a chord to be a pattern. This resulted in some notes being played syncopated whenever a chord was changed. With the more complex chord changes in particular the system would occasionally play just a note or two on the second beat, predicting that the chord would remain for longer than a measure. It predicted this because the chords lasted two beats each. The patterns that are discovered are small one chord patterns that reflected the chord not changing every beat. The performer could decide not to write music if the predicted chords are just an extension of the chords the performer is currently playing. The pattern matching could also prefer longer matches to short ones so that the two beat patterns aren't used as frequently.

The pattern discovery and matching suffers due to the realtime restriction. Many larger patterns are neglected in favor of realtime execution. Once the system finds any pattern in the music in which it hasn't discovered a pattern yet it updates the marker of where to look next. So if a pattern that was previously found (e.g. 'C' 'G') is then later found again, the marker, which keeps track of where to start looking for patterns next time, is updated to exclude the music containing the newly discovered pattern from future searches. This denies the system the opportunity to ever discover any larger form patterns. The irony here being that by saving one or two larger form patterns in a piece of music all of the smaller ones would be irrelevant as the larger form pattern would encompass all of the smaller ones excepting some cadential passages and variations.

Limiting the number of saved patterns to ten provides a disservice to the system at times. The advantage with only keeping track of the last ten patterns is that if a piece modulates to a different key, or if it changes progressions, the performer will be able to play along quite quickly because the newest patterns will be easily retrieved. The main problem is that all of the patterns that were relevant previously may be thrown out if the piece stays in the new key area for enough time. Often times pieces will move to a different key area, using new chord progressions and then move back to the original material. If all of the patterns were saved then when the piece moved back to the original material the performer would be able to play almost instantaneously.

Some problems definitely arise due to the implementation of key signature discovery. ‘F#’ which is not included in the major key list for ‘C’ major, can often arise as a chromatic passing tone, or as a part of a ‘D’ major chord being used to tonicize ‘G’. Tonicization is when a chord other than the tonic or root, or name of the key signature is temporarily emphasized as if it were the key signature. ‘A#’ can also be a chromatic passing tone which would exclude the possibility of ‘C’ major. ‘G#’ should be included in the ‘C’ major list as well. ‘G#’ is used in a ‘C’ major chord sometimes to move from a ‘C’ major chord to an a minor chord. The only note now not belonging to ‘C’ major would be ‘C#’ but this note also should be included because of Neapolitan 6th chords, used as a subdominant substitute.

If we were to include all of these possibilities then every key signature would have every note as a possibility meaning we would never know what key signature the piece is in. A secondary tool, allowing for a few instances of these less commonly used color tones, but not allowing for so many that ‘C’ major and ‘F#’ major are both being considered as possible key signatures after four or five bars

of music, merits development. A signal could be used that would allow for notes outside of the strict key signature to be played. The signal would work somewhat like a semaphore, having a value that gets decremented by the performance of a color tone, and incremented (up to a point) by a delayed computation.

The chordal analysis could be upgraded to include chords that do not function with just one pass over the piece, reading from left to right. Some chords, namely secondary dominants and the Neapolitan sixth function differently depending on where the music goes after the chord sounds. ‘D’ major, a secondary dominant in the key of ‘C’ major, could function as a chord used to tonicize ‘G’ major, or as a chord in a modulation (changing of key signature) to ‘G’ major.

The simple solution that was used to handle all of these issues was to neglect any of the logic needed to correctly interpret and analyse the music and instead only remember the last ten patterns that have been seen. In this way, if a piece modulates the performer is ‘confused’ for a moment and then with the new patterns it has accumulated, begins playing anew, in the new key signature.

The signal handler worked very well in solving implementation problems due to the reactivity in FrTime. Only one phrase would be performed instead of every phrase thanks to the value-function.

Chapter 7

Future Work

This work suggests several future projects. The most interesting and important work to be done would be to introduce the hybrid push-pull model into the FrTime language. This would complicate much of the logic and the semantics will have to be reconsidered just as it was done with push-pull hybrid [4]. The main point of interest here would be to compare the two models and see whether the efficiency changed drastically for particular problem sets.

Secondly, the analysis of the music being performed by the human player can be improved with some better logic from previous work done in melodic improvisation and recognition [3]. By recognizing melodies that the human plays, and relating them to specific chord changes, the performer could harmonize with the human player by predicting that the melody will return the next time that the chord progression comes around. This would also allow for some improved improvisation by manipulating melodies with transposition, retrograde, inversion and combinations of these manipulations. The static implementation of what is or is not harmonic to a key signature should be relaxed to provide chromatic notes and more color tones. This would require some overhauling of how key

signature is discovered.

With the pattern matching issues, a modified join function could be used to help put smaller patterns together and make larger ones [6]. This work would have to be modified, but an asynchronous joining behavior could run trying to put smaller patterns together to make larger ones which could then be permanently saved. This would solve more than one problem described in the results. With the addition of these larger form fragments the confusion of when thematic or original material returned after a period of time when the piece was developing new material would be remedied. The performer would know to look for larger forms if no small patterns matched. The behavior could also be triggered anytime the pattern list reached its full size.

The music value-functions could be improved with more complex models. For example, the rhythmically minded performer could pick passages that leave a distribution that approximates a normal curve.

The system itself can be improved to take advantage of other technologies that already exist. The system could accommodate multiple human players performing simultaneously. Multiple computer performers each performing with a different voice (e.g. saxophone, clarinet, etcetera) could play along with one or more human performers. The biggest roadblock to this work is finding an interface that can handle all of the MIDI traffic. Rewriting the system to have more than one performer does not require much more work: a controller module could handle this.

Another interesting avenue would be to make this framework so easy to use that non-programmers could program performers. This can be accomplished by using a very small, domain specific language that musicians would understand

given that they know some music theory. The musical value-functions require a higher level language, but with a few preprogrammed into this system already users could simply select different functions and decide which is best for the performer that they are writing.

In writing the performers, the user could be provided with a small library of useful functions for creating notes. In the scalar performer created here, two functions ‘get-next-note-in-scales’ and ‘get-prev-note-in-scales’ were created which given a list of possible key signatures and a note return the next, higher frequency, pitch in any of the provided scales or previous, lower frequency pitch in any of the scales. Other very simple functions could be written to create a library for the creation of performers.

The signal handler could also be used in many other domain specific applications like robotics and facial recognition.

Chapter 8

Conclusion

This work contributes a framework for writing personalized music improvisers that will react with the help of a functional reactive programming language, FrTime. The implementation in a reactive language created the problem of having more than one value for each point in time. A demand-driven signal is created which solves this problem and prevents excessive computation.

In Haskell, a lazy language, time functions are evaluated with a pull-based model and reactive values, like a mouse click, are evaluated with a push-based model. In FrTime, an eager language, everything is evaluated with a push-based model. The push-based model computes values for future time values up until that time value is the current time value. This is inefficient, adding overhead from function calls. The demand-driven signal minimizes function call overhead by being pull-based or demand-driven.

The results show that the system works in realtime with a human player. The music that the performers generate is dictated by both the computer performer and a music value-function. The music value-functions decide which phrase of

music the computer performer creates is chosen to be performed. Both of the performers created could benefit from more logic. The cadential performer should be less random. The scalar performer should not play notes from all possible scales. The music value-functions should be improved as well. A composition of the second and third music value-function deserves creation and experimentation.

The framework can be enhanced to improve both the amount of information saved and the type of information saved. A domain specific language for non-programmers which allows them to create musical performers merits creation. A push-pull hybrid reactive evaluation model needs to be tested against a pull-based model.

Bibliography

- [1] F. W. Burton. Indeterminate behavior with determinate semantics in parallel programs. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, FPCA '89, pages 340–346, New York, NY, USA, 1989. ACM.
- [2] G. H. Cooper and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *In European Symposium on Programming*, pages 294–308, 2006.
- [3] T. Crawford, C. S. Iliopoulos, and R. Raman. String matching techniques for musical similarity and melodic recognition. *Computing in Musicology*, 11:73–100, 1998.
- [4] C. Elliott. Push-pull functional reactive programming. In *Haskell Symposium*, 2009.
- [5] D. Gerhard. Computer music analysis. Technical report, 1997.
- [6] J.-L. Hsu, A. L. P. Chen, and H.-C. Chen. Finding approximate repeating patterns from sequence data. In *ISMIR*, 2004.
- [7] P. Hudak. Haskore music tutorial. In *In Second International School on Advanced Functional Programming*, pages 38–67. Springer Verlag, 1996.

- [8] J. lien Hsu, C. chin Liu, and A. L. P. Chen. Efficient repeating pattern finding in music databases. In *In Proceedings of the 1998 ACM 7th International Conference on Information and Knowledge Management*, pages 281–288, 1998.
- [9] M. Nelson and B. Thom. A survey of real-time midi performance, 2004.
- [10] B. Thom. Learning models for interactive melodic improvisation. In *Proceedings ICMC 99. 190193*, 1999.
- [11] Z. Wan, W. Taha, and P. Hudak. Real-time frp. *SIGPLAN Not.*, 36:146–156, October 2001.
- [12] Z. Wan, W. Taha, and P. Hudak. Event-driven frp. In *Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages, PADL '02*, pages 155–172, London, UK, 2002. Springer-Verlag.