CS290 Final Report
Virtual Piano Keyboard and Guitar Fretboard Widgets for Haskore

Harley Trung
Advisor: Paul Hudak

Spring 2009

Abstract

Functional Reactive Programming (FRP) is a programming paradigm for reactive programming using the building blocks of functional programming to model hybrid systems containing both continuous time-varying behaviors and discrete events. FRP has been applied to a variety of systems, including computer animation [EH97], robotics [HCNP02], and real-time systems [WTH01].

In computer music, we want to expose to the programmer the notion of discrete notes, events, and continuous audio signals, as well as the transition and interaction between these entities. The hybrid and dynamic nature of the problem makes FRP a good choice for modeling computer music systems [E09].

Haskore [HMGW96], a domain-specific language (DSL) built on top of Haskell, takes an algebraic approach to high-level music description and composition. Haskore has built-in functions to represent notes, rhythms, chords and many more. In Haskore a basic set of widgets is provided that are collectively referred to as the Graphical Music Interface (GMI) [H08].

However, Haskore GMI currently provides only a limited set of UI widgets that can only create buttons, value sliders, checkboxes, radio buttons, text displays, and allow MIDI input and output. This project sets out to add to Haskore two high-level, sophisticated widgets: a piano keyboard and a guitar fretboard. There are four goals in mind. First, the design and implementation should be as generic as possible, maximizing common features to be shared by a piano keyboard and a guitar fretboard. Second, the design should reflect real life instruments. For example, the virtual piano keyboard should behave almost identical to a MIDI keyboard that can be plugged into a computer for MIDI input; a pedal can be attached to it and there are nobs to change various settings such as echo, transpose, etc. Third, the interface must not only be

simple enough to customize and use, but also support a rich set of features such as echo, transpose, pedal, import-song, change-instrument and MIDI output. Finally, I discuss the challenges encountered during the development and potential contributions of these widgets to a Haskore user.

1 Introduction

The initial project proposal was to extend the functional UI library in Haskore by integrating the GMI with a popular GUI framework so as to allow direct porting of UI widgets to GMI. However, there is no current GUI framework written in Haskell that is stable enough to use. For example, Phooey, a popular functional UI library for Haskell, uses wxHaskell to provide an FRP-like abstraction for GUI programming. However, wxHaskell itself is built on top of wxWidgets, a C++ library, and is having some known issues in Linux. I encountered problems testing out Phooey and many of the known bugs are not yet solved. Other Haskell GUI frameworks such as Grapefruit, wxFruit, gtk2hs are also new and unstable. Moreover, these external GUI packages are very different from Haskore. Therefore, the goal of integrating Haskore's Functional UI with one of these frameworks while maintaining the signal level abstraction is too ambitious. The project was therefore limited to designing instrument widgets for Haskore.

The project is still interesting and challenging nevertheless. Haskell is different from conventional imperative and object-oriented languages such as C, C++, Java, C# and so on. It requires a different mind-set to program in Haskell, with the goals towards the purity, elegance in the programs.

For this report to be more readable, I will first explain some aspects of Haskore, mainly borrowing from [H08].

2 A brief introduction of Haskore's GMI

Haskore's Graphical Music Interface contains a basic set of widgets. There are two levels of abstraction: At the *user interface* (UI) level, basic IO-like commands are provided for creating graphical sliders, push-buttons, and so on for input, and textual displays and graphic images for output. In addition to these graphical widgets, the UI level also provides an interface to standard Midi input and output devices. A virtual keyboard or fretboard belongs to the graphic input widget category.

The second level of abstraction of the GMI is the *signal level*. A signal is a time-varying quantity that captures the behavior of many widgets. A special case of a signal is an event,

such as a MIDI event, which occurs each time a Note-On or Note-Off message is sent to output.

2.1 Signals

Abstractly, signal can be *thought of* as

Signal a = Time →a

where Time is represented as a Double in Haskore. It is helpful to think of signals like the following (Note that it is not how Haskore defines signal, but the idea is there)

data Signal a = Signal (Time →a)

From here on our discussion will mainly be on actual Haskell type signatures. An event can then simply be defined as a value of type *Signal (Maybe a)* where

data *Maybe a = Nothing | Just a*

Therefore, an event is type synonym:

type *EventS a = Signal (Maybe a)*

2.2 The UI Monad

GMI widgets are created at the UI level. Similar to IO, UI is an abstract type and is a monad, which allows us to use the **do** syntax. See the table for the current list of GMI Input Widgets. The names and type signatures should suggest their functionality.

label :: String → UI ()
display :: Signal String → I ()
button :: String → UI (Signal Bool)
checkbox :: String → Bool → UI (Signal Bool)
radio :: [String] → Int → UI (Signal Int)
hSlider , vSlider :: (RealFrac a) → (a, a) → a → UI (Signal a)
hiSlider , viSlider :: (Integral a) → a → (a, a) → a → UI (Signal a)
canvas :: Dimension → EventS Graphic → UI ()

2.3 MIDI Input and Output

midiIn :: Signal DeviceID → UI (EventS [MidiMessage])
midiOut :: Signal DeviceID → EventS [MidiMessage] → UI ()

midiOut takes a stream of MidiMessage events and sends to MIDI output device, and *midiIn* generates a stream of MidiMessage events corresponding to the message sent by the Midi input device.

A virtual keyboard or fretboard is an input device, therefore we'd like it to returns the same type as *midiIn*. It may take more arguments depending on the specifics of the instruments, but it must return a stream of MidiMessage EventS. *EventS MidiMessage* should work but because the type accepted by *midiOut* in Haskore is *EventS [MidiMessage],* I will stick to that when I define the virtual keyboard.

From now on we are using the following synonym:

type *EMM = EventS [MidiMessage]*

2.4 Keyboard and fretboard as input devices

We aim to have the piano keyboard and guitar fretboard share as many common characteristics as possible. Acting as MIDI input devices, the functions piano and guitar that define their corresponding widgets should have the return type UI EMM, just like the widget *midiIn* above. For example, here is a sneak at the type signatures of the two widgets.

piano :: PianoKeyMap → Midi.Channel →Signal InstrumentData →EMM →UI EMM
guitar :: GuitarKeyMap→ Midi.Channel →Signal InstrumentData →EMM →UI EMM

As one may have noticed, *piano* and *guitar* also allows *EMM* as input. This is analogous to a real keyboard that has a MIDI-in port that propagates MIDI messages. We would also like to define a piano or guitar on a specific MIDI channel. This way we can call multiple piano/guitar widgets that output MIDI messages on different channels so that they can be combined and played simultaneously.

3 Common interface of the keyboard and fretboard

3.1 A key widget should be independent of other key widgets

In this section, we mainly take examples from the piano keyboard but the concept is similar on the guitar fretboard. Any difference is highlighted in the later section.

On many music instruments, a sound is made when a key is pressed. The sound corresponds to a certain music pitch. In our widget, we also need to animate that a key is being pressed or released. A key on a piano keyboard should be animated as pressed in three ways: when the user press a key on the computer keyboard that is mapped to the piano keyboard, when the user's mouse clicks on that key on the screen, or when one of the musical notes of the song being played at this particular point in time is the exact same as the note (in terms of absolute pitch) mapped to that key. We define a *KeyBool* data type to reflect this:

data *KeyBool = KeyBool {keypad: Bool, mouse: Bool, song::Bool}* ***deriving*** *(Show, Eq)*
isKeyDown (KeyBool False False False) = False
isKeyDown _ = True
isKeyPlay (KeyBool False False _) = False
isKeyPlay _ = True

isKeyDown is a function that checks if a key should be animated as pressed or unpressed. *isKeyPlay* checks if a sound (MidiMessage NoteOn) should be sent according to the interaction with the piano keyboard. The sound from a song being played is treated separately. This is just a design choice: when a note played in the song is out of the range of the piano keyboard, no key will be seen as pressed, but we still want the sound output to go through.

A key can be thought of as a more basic "widget" itself. A key is pressed or unpressed independently of others. Should the key produce the MIDI message directly? For better modularity, I keep the sound interpretation outside the key because it alleviates the key from the responsibility of knowing which MIDI channel to write to and what musical notation it associates with.

Therefore, a *piano key widget* only need to know the (computer) key binding (*Char*), how to draw itself (*KeyType*), any time-varying information such as whether it is being pressed by a song event and what music notation it should display (*Signal KeyData*). The widget outputs a continuous signal about its state which is encapsulated in the type *Signal KeyBool*

data *KeyData = KeyData { pressed :: Maybe Bool, notation :: Maybe String, offset :: Int } deriving (Show, Eq)*

mkKey :: Char → KeyType →Signal KeyData →UI (Signal KeyBool)

The same is idea is extended to a *guitar key widget*, which represents a string being pressed at a certain fret on the guitar neck.

3.2 Meta-key information

We also need to keep track of instrument-wide properties and allow it to be time-varying

data *InstrumentData = InstrumentData { showNotation::Bool, keyPairs :: Maybe [(AbsPitch, Bool), transpose :: AbsPitch]}*

3.3 Take keys to sound

The intermediate level between the representation of a piano key being pressed and a sound being played is the musical notation associated with that key. A key sends a continuous signal of *KeyBool* type. However, a MIDI message is of event type: discrete events of NoteOn or NoteOff messages. It is not desirable to continuously send a NoteOn signal, for example. We take advantage of the function *unique* provided by Haskore for dealing with signal:

unique :: Eq a => Signal a -> EventS a

We define a *mkKeys* widget that converts the continuous signal of the state of each key (*Signal KeyBool*) to events that happen whenever that signal changes (which means the key is pressed or release) and map the events to the corresponding musical notation AbsPitch. (An absolute pitch is an integer that identify a pitch and it also corresponds to the Key data type in a MIDI Message.)

mkKeys need to know the mapping of the keys to their corresponding *KeyType* and *Absolute Pitch*. It returns an event stream of pairs that identify if a pitch should be played or stopped. *mkKeys* also need to create and send key-specific *Signal KeyData* to each key. To recap: *Signal KeyData* allows us to pack extra time-varying information sent to each key: for example if a notation should be displayed (depending on the user's toggling of the notation checkbox) or if that key should be pressed as its note is being played in a song. It is the breakdown of *Signal InstrumentData* for individual keys.

mkKeys :: [(Char, KeyType, AbsPitch)]→Signal InstrumentData→UI (EventS [(AbsPitch, Bool)])

We will map a function *getKeyData* that create *KeyData* for each key based on the absolute pitch value

getKeyData :: AbsPitch →Signal InstrumentData →Signal KeyData

Next, we just need a function that writes to EMM for us - the MIDI Input format. It is easy to write the following and lift it to the level of events by using the *(=> >)* function

pairToMsg :: Midi.Channel -> [(AbsPitch, Bool)] -> [MidiMessage]

Note the (=> >) function is from Haskore's Signal library:

(=> >) :: EventS a → (a → b) → EventS b

First, let us review the type of *MidiMessage* in Haskore. Note that Haskore uses module Codec.Midi which interfaces with MIDI files and also defines Message as a standard MIDI message. MidiMessage adds the notation of ANote about which we are not concerned for now.

– from module Codec.Midi
data *Message* =
– Channel Messages
NoteOff {channel :: !Channel , key :: !Key, velocity :: !Velocity }
| *NoteOn{channel :: !Channel , key :: !Key, velocity :: !Velocity }*
| *ProgramChange {channel :: !Channel , preset :: !Preset }*
| ...
– Meta Messages
| *TempoChange ! Tempo* |
| ...
deriving *(Show , Eq)*

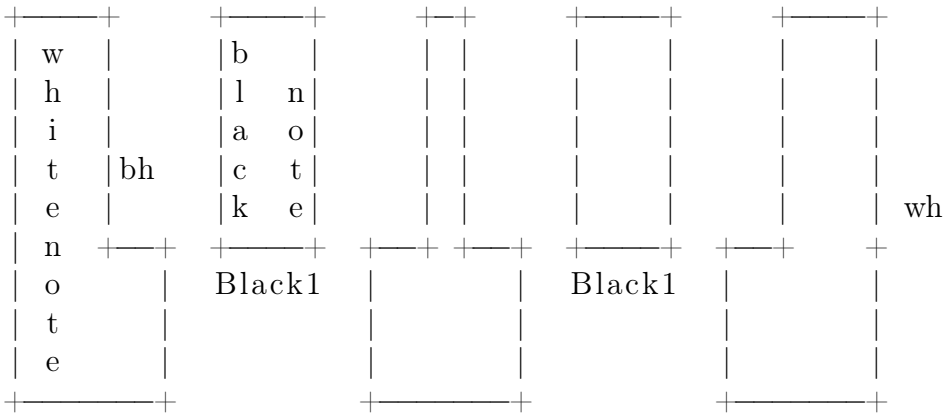
– from Haskore
data *MidiMessage = ANote{channel :: Channel , key :: Key,*
velocity :: Velocity, duration :: Time }
| Std Message
deriving *Show*

Only NoteOff and NoteOn messages are needed because they correspond to a key being pressed or release.

4 The Virtual Piano Keyboard

4.1 KeyType

As we all know, a piano keyboard has white keys and black keys. All black keys look identical, but white keys take three different shapes. Take the first 5 notes of a keyboard starting from C and separate them we associate the KeyType with shapes



White1 White2 White3

The piano keyboard layout repeats for each octave so it is not difficult to prepare our *defaultKeyLayout*

data *KeyType = White1 | White2 | White3 | Black1* **deriving** *(Show, Eq)*
defaultKeyLayout = cycle *[White1, Black1, White2, Black1, White3, White1, Black1, White2, Black1, White2, Black1, White3]*

The layout facilitates the natural drawing of the keys from left to right in the order above, allowing each key to draw itself (independently without the need to know about other keys' position) where the current *cursor* is and at the end leaves the cursor where the next key can draw.

Apart from drawing routines, we also need an auxiliary function to check if a point (x-y coordinates) is inside one of the above shapes or not (so as to decide whether a key is pressed when the mouse is clicked or unclicked)

insideKey :: KeyType →Point →BoundingBox →Bool

5 Virtual Guitar Fretboard

5.1 KeyType

Design a virtual guitar is more challenging in a few ways. First, the abstraction we have so far seem to be more suitable between a key and a string on which it is pressed. Hence a key is the an intersection between a string and a fret. A standard guitar has 6 strings, which seem to behave like 6 keyboard above.

But on each string, no matter what key is pressed, a sound is only made when the string is picked. In addition, a key pressed on the right of another key (on the same string) makes that key have no effect. In other words, only the right-most pressed key produces a sound when the right string is picked.

Guitar frets do not vary much in shape, except for the width which decreases in those that are further on the right. Therefore we only need a fret count number in order to know how to draw it:

type *KeyType = Int*

For aesthetics, we add a function to draw the guitar head before drawing the strings. It needs to know how many strings there are:

drawHead :: Int → UI ()

The *guitar key widget* only differs from the *piano key widget* in the drawing functions used.

5.2 Guitar String

Unlike in the virtual piano keyboard, there is more to do with a guitar string rather than just manage a set of keys with *mkKeys*. First we need to define the plucking of a string:

pluckString :: Char →UI (Signal Bool)

pluckString is similar to a simplified key widget: it checks if a certain character (which is the only argument it takes) and respond with the familiar *Signal Bool*

We also need to modify *mkKeys* to handle the check when a string is plucked without any key pressed (free string plucking). Note the addition of the first two arguments which are the pitch of the free string and whether it's plucked:

mkKeys :: AbsPitch -> Signal Bool -> [(Char, KeyType, AbsPitch)] -> Signal Instrument-Data -> UI (EventS [(AbsPitch, Bool)])

Now *mkString* is a very similar to *mkKeys* in the piano except for the first argument which specifies the key-binding for the string plucking.

mkString :: ([Char], Pitch, Char) -> Signal InstrumentData -> UI (EventS [(AbsPitch, Bool)])

6 Add-on widgets

We have looked at the design and implementation of two virtual instruments: keyboard and guitar. The main difference between our virtual instrument and the one in real life is that we leave out the extra features such as pedal, volume control, transpose setting, playing a predefined song etc. This section illustrates that such do not belong to the instrument at its core. Some can be achieved by post-processing the MIDI output that comes out of the instrument. Our virtual instrument allows input that changes its behavior such as displaying a notation, transpose, or it can also read from a song and highlights the notes that are played on the keyboard or fretboard.

The greatest advantage of our approach so far is that all these widgets are shared by both the guitar and the piano and can be generic to a new virtual instrument designed this way.

6.1 Non-intrusive Widgets

6.1.1 Instrument Select Widget

This is the simplest widget of all. It takes advantage of the power of the meta message syntax in a MIDI message. It displays the list of MIDI instruments available, provides a slider, and add to EMM a message that change the instrument on a certain channel whenever the user move the slider to choose a different instrument. The function takes MIDI channel and the default instrument number, the the event stream of MIDI messages as arguments.

selectInstrument :: Midi.Channel →Int →EMM →UI EMM

6.1.2 Song Widget

Song widget takes a list of songs as input, provides the user with a list of songs to choose and sends out an event stream of MidiMessage (EMM) as soon as the user click Play on a selected song.

selectSong :: [Music Pitch] →UI EMM

Note *Music Pitch* is the standard data type to represent a melody in Haskore.

6.1.3 Echo

Echo widget is another post-process one: it takes EMM and recursively add decaying sound to it whenever the Echo checkbox is ticked. It also provides a slider to adjust decay rate and echo frequency at real time. Its type signature is simple

addEcho :: EMM → UI EMM

6.2 Intrusive Widgets

6.2.1 Notation Display Widget

This widget differs from the ones so far in the way that it does not modify any content of the MidiMessage. It allows real time modification of the InstrumentData so that the user can toggle the display of notation on the keys or not. Because toggling the notation display

can be quite common, the checkbox and be checked or unchecked via a keyboard shortcut. The key-binding is specified in the first argument.

addNotation :: Char→Signal InstrumentData →UI (Signal InstrumentData)

This is particularly useful to keep track of the pitch of the keys after the user performs some transpose on the instrument, which brings us to the Transpose widget.

6.2.2 Transpose Widget

Transpose can be done at the MidiMessage level and thus can be independent of the instrument itself, just like most of the previous widgets (except *addNotation*) However this way would cause a discrepancy with the notation display when it is toggled on. Therefore, similar to toggling notation display, we update the real-time changes to *Signal InstrumentData* which is then passed into the instrument.

addTranspose :: Signal InstrumentData→UI (Signal InstrumentData)

6.2.3 Pedal Widget

There are two possibilities in designing a pedal widget. One that one manipulate the MIDI message and one that changes modify the internal state of the instrument. The latter is more intrusive but easier to implement. Due to the time limit, I make one exception to be in favor of implementation over intrusion-free style.

Similar to *addNotation*, the pedal widget takes a key binding for easier toggling of the pedal and modifies *Signal InstrumentData* in real time.

addPedal :: Char →Signal InstrumentData →Signal InstrumentData

6.3 Other possible add-ons

6.3.1 Generic and non-intrusive widgets

The presence of the above non-intrusive widgets illustrate that the user can write his own widget that modifies the MidiMessage and does not care about the inside of the virtual instrument. Examples of easy-to-implement widgets in this category are:

1. A Global Transpose widget that transpose the pitch of all notes in the MidiMessage

2. A Volume widget that changes the velocity component in the NoteOn and NoteOff message. This can be useful when the user has multiple instruments loaded on the screen and would like to have one particular keyboard or guitar to sound louder than the rest. This widget can be applied to a single or a set of virtual instruments.

6.3.2 Instrument-specific widget

In our implementation, the guitar fretboard is inferior to the keyboard in some ways due to the awkward key-binding with the computer keyboard and most importantly, the very limited support for concurrent keypresses from the computer keyboard. One can write guitar-specific widgets to make it more fun to play with the virtual guitar

1. A chord playing widget that allows the user to hold down to a chord name and press UP key or DOWN key to strum up or down. For example, hold 'D' and press UP to strum a D Major chord from string #6 up to string #1; hold 'd' and press DOWN to strum a D minor chord from string #1 down to string #6. It becomes more challenging but one can even add key-binding UP+'3' to strum from string #3 up to string #1. Due to multiple ways of baring a D major or D minor, the widget may be designed to take the preferred chord binding in the arguments. For example, a pair ('D', "x00232") would mean that D Major should be played with string #1 pressed at 2nd fret, string #2 at 3rd fret, string #3 at 2nd fret, string #4 and $5 free, and string #6 must be muted during strumming.

6.3.3 Load/Save configuration widget

Some of the magic performed by our virtual instrument relies on the *Signal InstrumentData* parameter. The keyboard or fretboard widgets are called from an UI monad which allows IO actions. *saveConfig* widget allows *Signal InstrumentData* variable to pipe through intact but the widget saves the information to a file.

saveConfig, loadConfig :: FileName→Signal InstrumentData →UI (Signal InstrumentData)

We include *FileName* as a string in the argument because GMI does not have a stable support for textbox so that the user can enter text yet. Conceptually, the file name may be entered by the user at run time.

7 Usage and Demo

Let us review the type signatures of our virtual instruments

piano :: PianoKeyMap → Midi.Channel →Signal InstrumentData →EMM →UI EMM

guitar :: GuitarKeyMap→ Midi.Channel →Signal InstrumentData →EMM →UI EMM

To allow much flexibility and customizability, these type signatures are not ideally short. However, default parameters are provided to make the setup of these instruments painless.

For the piano keyboard:

type *PianoKeyMap = ([Char], Pitch)*
defaultMap1, defaultMap2 :: PianoKeyMap
defaultMap1 = ("q2w3er5t6y7uQ@W#ERT^Y&U", (C,3))*
defaultMap2 = ("zsxdcvgbhnjmZSXDCVGBHNJM", (C,4))

For the guitar fretboard:

type *GuitarKeyMap = [([Char], Pitch, Char)]*
defaultMap :: GuitarKeyMap
defaultMap = [string1, string2, string3, string4, string5, string6]
string6 = ("1qaz__________", (E,6), '\b')
string5 = ("2wsx__________", (B,5), '=')
string4 = ("3edc__________", (G,5), '-')
string3 = ("4rfv__________", (D,5), '0')
string2 = ("5tgb__________", (A,4), '9')
string1 = ("6yhn__________", (D,4), '8')

Common default variables for these instruments:

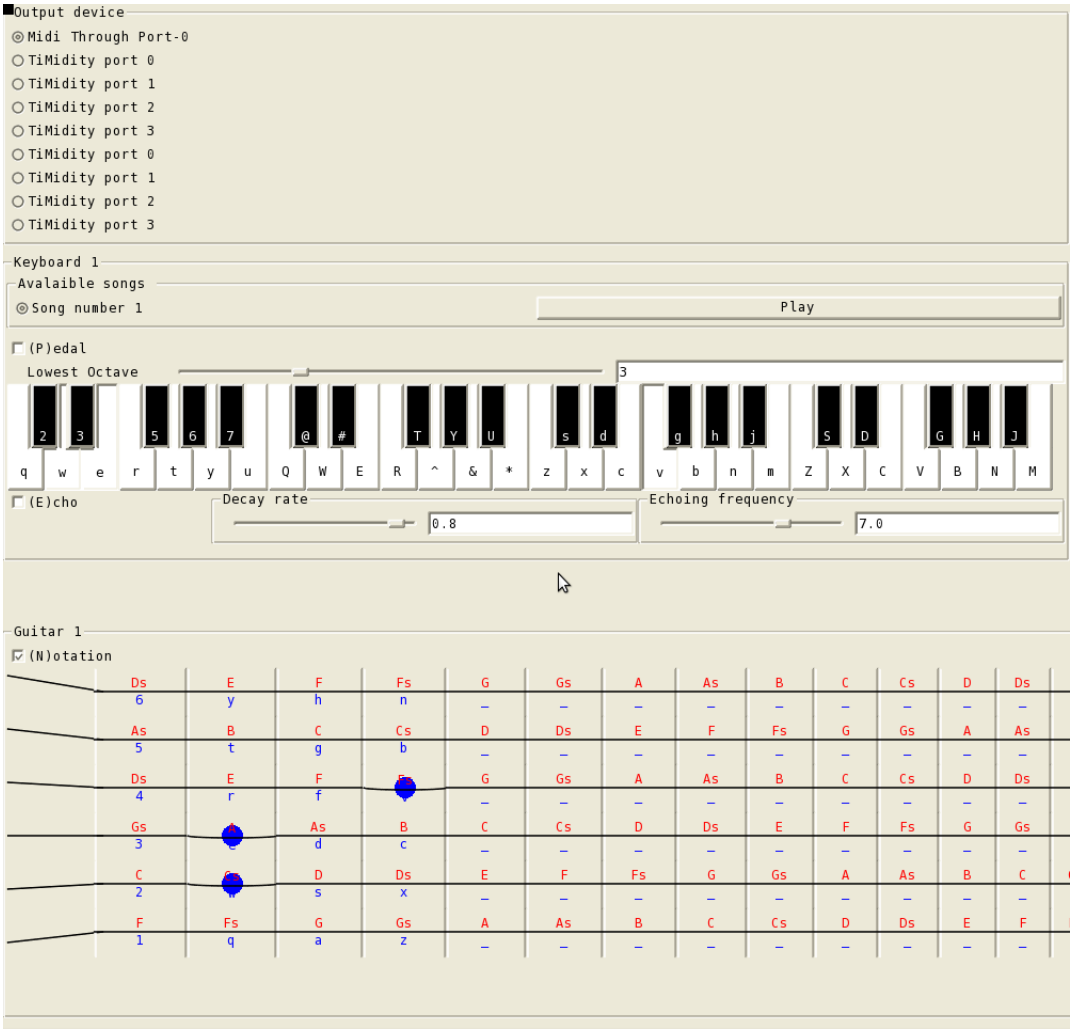
nullEMM = constant Nothing
defaultInstrumentData = constant $ InstrumentData False Nothing 0 False

7.0.4 Example:

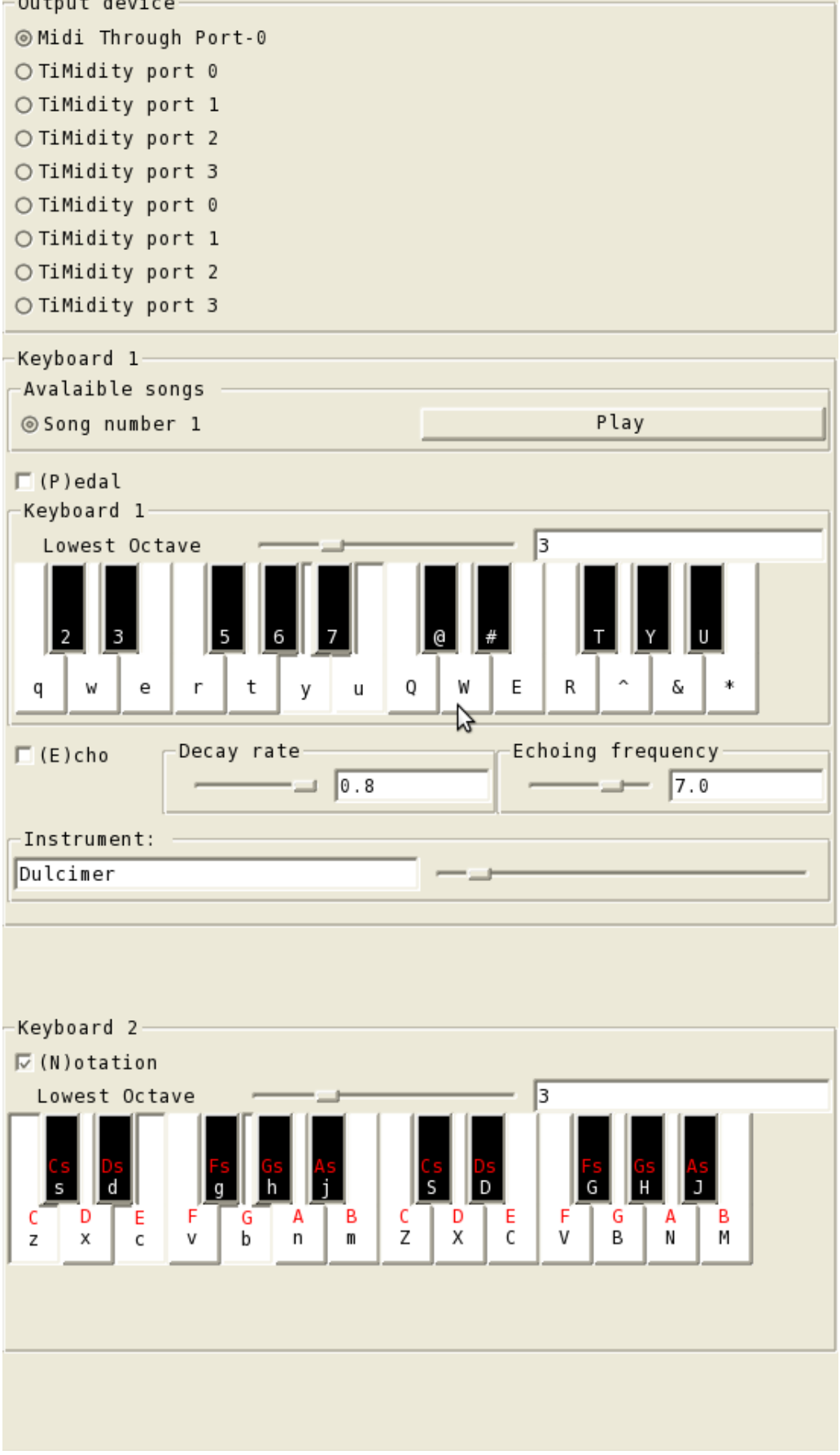
Below is the code and the screen-shots of the results

```
module Demo where

import InstrumentBase
import Piano
```

```
import Guitar
import Helper
import UI

demo1 = runUIEx (1000,1200) "Demo Virtual Instruments" $ do
      devId   <- selectOutput

      piano1 <- title "Keyboard 1" $ do
        songEMM <- selectSong [fjfj]
        pedalData <- addPedal 'P' defaultInstrumentData
        msg <- piano defaultMap0 0 pedalData songEMM
        addEcho msg

      guitar1 <- pad (0,50,0,0) $ title "Guitar 1" $ do
        notationData <- addNotation 'N' defaultInstrumentData
        guitar [string1, string2, string3, string4, string5, string6] 1 notat

      midiOut devId (piano1 |+| guitar1)

demo2 = runUIEx (550,1000) "Demo 2 keyboards" $ do
      devId   <- selectOutput

      piano1 <- title "Keyboard 1" $ do
        songEMM <- selectSong [fjfj]
        pedalData <- addPedal 'P' defaultInstrumentData
        msg <- title "Keyboard 1" $ piano defaultMap1 0 pedalData songEMM
        msgWithEcho <- addEcho msg
        selectInstrument 1 15 msgWithEcho

      piano2  <- pad (0,50,0,0) $ title "Keyboard 2" $ do
        notationData <- addNotation 'N' defaultInstrumentData
        piano defaultMap2 1 notationData nullEMM

      midiOut devId (piano1 |+| piano2)
```


Output device
Midi Through Port-0
TiMidity port 0
TiMidity port 1
TiMidity port 2
TiMidity port 3
TiMidity port 0
TiMidity port 1
TiMidity port 2
TiMidity port 3
Keyboard 1
Avalaible songs
Song number 1
Play
(P)edal
Lowest Octave
3
(E)cho
Decay rate
0.8
Echoing frequency
7.0
Guitar 1
(N)otation

Demo2:


Output device
Midi Through Port-0
TiMidity port 0
TiMidity port 1
TiMidity port 2
TiMidity port 3
TiMidity port 0
TiMidity port 1
TiMidity port 2
TiMidity port 3
Keyboard 1
Avalaible songs
Song number 1
Play
(P)edal
Keyboard 1
Lowest Octave
3
2 3 5 6 7 @ # T Y U
q w e r t y u Q W E R ^ & *
(E)cho
Decay rate
0.8
Echoing frequency
7.0
Instrument:
Dulcimer
Keyboard 2
(N)otation
Lowest Octave
3
Cs Ds Fs Gs As Cs Ds Fs Gs As
s d g h j S D G H J
C D E F G A B C D E F G A B
z x c v b n m Z X C V B N M

References

[E09] Eric Cheng. A Purely Functional Interactive Computer Music Framework

[EH97] Conal Elliott and Paul Hudak. Functional reactive animation. In International Conference on Functional Programming , 1997.

[HCNP02] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. In Johan Jeuring and Simon L. Peyton Jones, editors, Advanced Functional Programming , volume 2638 of Lecture Notes in Computer Science , pages 159–187. Springer, 2002.

[WTH01] Z. Wan, W. Taha, and P. Hudak. Real-time FRP. In International Conference on Functional Programming , Florence, Italy, September 2001

[HMGW96] Paul Hudak, Tom Makucevich, Syam Gadde, and Bo Whong. Haskore music notation - an algebra of music. Journal of Functional Programming, 6(3):465–483, 1996.

[H00] Paul Hudak. The Haskell School of Expression: Learning Functional Programming through Multimedia. Cambridge University Press, New York 2000

[H08] Paul Hudak. The Haskell School of Music. Yale University, 2008