An Interface for Realtime Music Using Interpreted Haskell

Renick Bell
independent researcher
Chuo-ku, Shinkawa, 2-27-4-2414
Tokyo, 104-0033
Japan
renick@gmail.com

Abstract

Graphical sequencers have limits in their use as live performance tools. It is hypothesized that those limits can be ovecome through live coding or text-based interfaces. Using a general purpose programming language has advantages over that of a domain-specific language. However, a barrier for a musician wanting to use a general purpose language for computer music has been the lack of high-level music-specific abstractions designed for realtime manipulation, such as those for time. A library for Haskell was developed to give computer musicians a high-level interface for a heterogenous output enviroment.

Keywords

live coding, realtime performance, Haskell, text-based interface

1 Introduction

In this paper, a usability problem of live computer music will be briefly examined, and the solution of using a general purpose programming language as a shell for music will be presented. The necessary components created by other developers which were used will be introduced. A library called Conductive[1], developed to make a Haskell interpreter into such a shell, will then be described in some detail. A conceptual example and some actual code examples will be presented. Finally, conclusions reached through the development of this library will be presented.

2 The Problem

Graphical sequencers are poor tools for live musical performance in the judgement of this author. Users interact with them primarily through a mouse or a limited number of keyboard shortcuts and allow limited customizations to the manner in which they are controlled. Previous experiences with GUI tools in performance showed them to be inflexible and awkward when trying to execute complex sets of parameter changes simultaneously.

This problem is exacerbated when considering the wide variety of synths which exist. Musicians would like to use them together freely, but coordinating them is difficult. For use with graphical sequencers, synths employ separate GUIs which are almost always point-and-click and thus cannot be easily manipulated simultaneously with other parameters.

One possible solution to this problem may be the use of a programming language as a tool for live coding of music or as a text-based interface [Collins et al., 2003]. A general purpose programming language which has abstractions for music can serve as a control center or shell for a heterogeneous set of outputs. In text, the user can write out complex parameter changes and then execute them simultaneously. A wide variety of existing programming tools and text-manipulation utilities can be used to make this process more efficient.

Computer music languages, such as SuperCollider [McCartney, 2010] and Chuck [Wang, 2004] exist. McCartney, the creator of SuperCollider, says a specialized language for computer music isn't necessary, but general purpose programming languages aren't ready yet practically [Mccartney, 2002]. Musicians manage to make music with domain-specific tools, but those are unsatisfactory in many ways, such as lack of libraries and development tools and slow performance.

General purpose programming languages have libraries for dealing with music, but they are particularly limited in number regarding real-time music systems. Some which already have such capabilities through libraries include Scheme through Impromptu [Sorensen and Brown, 2007] or Lua through LuaAV [Wakefield and Smith, 2007].

The Haskell programming language was seen as a good candidate because of several factors:

[1] http://www.renickbell.net/conductive/

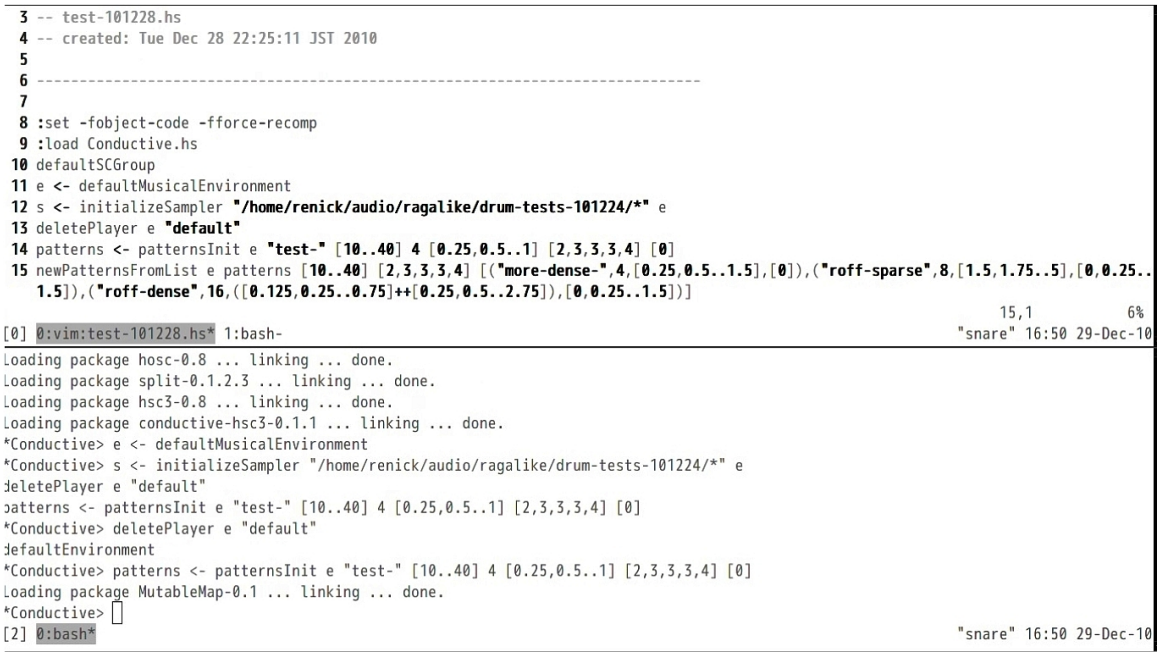

Figure 1: A screenshot of Conductive in use

expressivity, speed, static type system, large number of libraries, and ability to be either interpreted or compiled. It lacked a library suitable for this author for realtime manipulation of musical processes. McClean is also developing Tidal [McLean and Wiggins, 2010], a Haskell library with a similar aim.

3 The Solution

A Haskell library called Conductive was created. It contains abstractions for musical time, musical events, and event loops. This gives the Haskell interpreter the ability to function as a code-based realtime sequencer for any output targets which can be connected to the system. Conductive does not aim to be an audio language, but a controller for audio output targets. The user is free to choose any OSC-aware output target, and this library is proposed as a way to coordinate those outputs. Another way to think of it is as a shell or scripting environment for realtime music.

A library for getting, composing, and sending messages to JackMiniMix, an OSC-based mixer for JACK developed by Nicholas Humfrey [Humfrey, 2005], was created[2].

A simple terminal-based clock visualization was also created.

[2] http://www.renickbell.net/doku.php?id=jackminimix

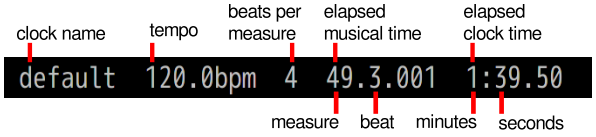

Figure 2: The terminal clock display

4 Utilized Tools from Other Developers

Before explaining the details of Conductive, it is necessary to list the components it was integrated with. The Glasgow Haskell Compiler Interpreter (ghci) [Peyton Jones et al., 1992] was the core component used for executing Haskell code. Code was composed in vim [Moolenaar, 2011], and sent to ghc via the tslime plugin [Coutinho, 2010]. For OSC communication, Rohan Drape's hosc library was used [Drape, 2010]. Output targets used were scsynth, the synthesizer component of SuperCollider [McCartney, 2010], and JackMiniMix. Drape provides a library for communicating with scsynth via OSC called hsc3 [Drape, 2009].

5 Conductive in Detail

5.1 Overview

This library exists to wrap concurrent process manipulation in a way that makes controlling their timing more intuitive for musicians. At the same time, the library aims at being as concise as possible to lessen the burden on the user.

The core components of the library are the data structures Player and MusicalEnvironment and a set of functions using these data structures. A user designs a set of functions carrying out musical actions, such as playing a note on a synth or adjusting the parameter of synth. The user defines TempoClocks which have a tempo and time signature. The user also defines functions, called IOI (interonset interval) functions, describing how long to wait between executions of actions. These functions are stored in a MusicalEnvironment. A Player is a collection of one action function, one IOI function, and one TempoClock and other related data. A Player is put into an event loop in which actions are executed after every defined IOI by using the play function.

Conceptually, it has similarities with the concepts in SuperCollider of Routines, Tasks, and Patterns. Some key similarities and differences are noted below, along with details on each of these components.

5.2 TempoClock

The tempo is part of a TempoClock, a concept from SuperCollider which is reimplemented here in Haskell. A TempoClock is like a metronome keeping the current tempo but also containing information about time signature and when tempo or time signature has been changed.

A TempoClock is a record of the time the clock was started, a list of TempoChanges, and a list of TimeSignature changes. This allows a user to independently manipulate both tempo and time signature and to use these for composing and performance in addition to regular POSIX time.

TempoClocks are stored in the MusicalEnvironment.

5.3 Players

A data structure called a Player was designed as a way to sequence IO actions. Players contain references to particular data which is stored in the MusicalEnvironment. The collection of data referenced by the Player results in a series of actions being produced once the Player is played. This data consists of:

- the name of the Player
- its status (stopped, playing, pausing, paused, stopping, resetting)
- a counter of how many times it has run an action

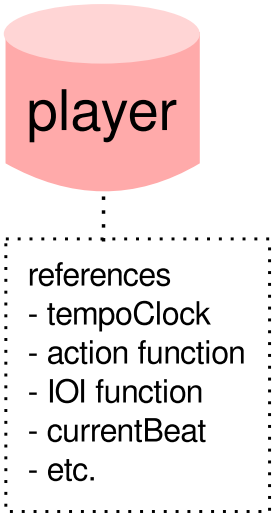

Figure 3: Player: a data structure filled with references

- which clock it is following
- which IOI function it is using
- which action function it is using
- which interrupt function it is using
- which beat its next action occurs on
- which beat it started on
- the POSIX time at which it was last paused

An action function is a function that describes an event. An action function outputs a value of the IO unit type. This basically means some kind of side effect is produced without returning a value like a double or a string. In practical terms, this could be a synthesis event, a parameter change, or the action of playing, pausing, or stopping other Players or itself. It is thought that the user would use functions which send OSC messages to connected OSC-aware applications. The action named in the Player can only take two parameters: the Player triggering the action and the MusicalEnvironment it should read from. Beyond that, the design of the action is left to the user. A user might prefer to have many Players with simple actions, a few Players with complex actions, or some other combination.

A fundamental concept is that of the time interval between the start times of two events, or interonset interval (IOI). [Parncutt, 1994] SuperCollider refers to this as “delta” with regard to Patterns or “wait” for Routines. The IOI is defined in beats, and the actual time between events is calculated using the IOI value and the

TempoClock referenced by the Player it is associated with. IOI functions should also be designed to read the data from a Player and a MusicalEnvironment. They can be designed in any way the user desires, including always returning a particular value, stepping through a list of values stored in a list somewhere, randomly choosing a value, or anything else the composer can imagine.

An interrupt function is a function which is run once every time the play loop runs. It is useful for debugging purposes, and may be used to trigger other actions, such as stopping the player on a condition.

Players bear some resemblance to Tasks or Patterns in SuperCollider; they can be played, paused, and stopped to produce music. However, while Patterns in sclang can produce streams of any kind of data, Players in Conductive are designed to produce streams of side effects. While the data in a Pbind in SuperCollider is generally private [Harkins, 2009], all the data contained by a Player is visible.

Players are stored in the Player store, a mutable key-value store where the keys are Player name strings and the values are the Players themselves. This in turn is part of the MusicalEnvironment. How patterns are stored in SuperCollider is up to the individual user. This library provides a readymade structure for that purpose.

5.4 MusicalEnvironment

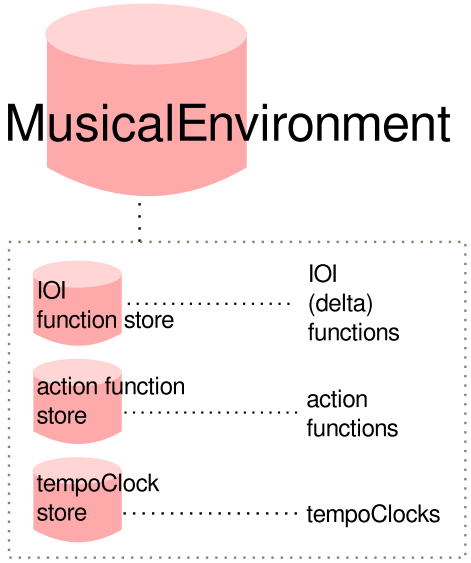

Figure 4: MusicalEnvironment: a data structure for storage

The MusicalEnvironment is a place to store data which is used by any of the user-initiated event loops. This data consists of:

- the name of the environment
- a store of Players
- a store of TempoClocks
- a store of IOI functions
- a store of action functions
- a store of interrupt functions

5.5 Play

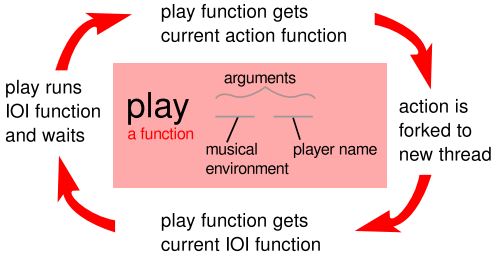

Figure 5: The play event loop

The play function starts a thread which forks other processes according to a schedule determined by the IOI function referenced in the Player. It takes a MusicalEnvironment, a Player store, and a Player name as arguments. First, the play function checks which action is referenced in the Player. It retrieves that function from the MusicalEnvironment and forks it to a thread. It then checks which IOI function is referenced in the Player. It runs that function and receives a numeric value specifying how long to wait in terms of beats. It then corrects that amount for jitter and sleeps for the corrected length of time. When the thread wakes up, the loop — checking the action and so on — repeats.

It produces roughly similar results to calling play on a Pattern in SuperCollider in that it begins a process; however it is structured differently.

The problem of dealing with the delays in scheduled events is significant. Because various processes, including garbage collection, can conceivably interfere with correct timing, correction of jitter is included in the play event loop. This library does not introduce a novel method for managing such delay, but rather adopts a design from McLean [McLean, 2004]. An event intended to occur at time x actually occurs at time x + y, where y is the amount of time by

which the event is late. The next event is scheduled to occur at time x + z, where z is the IOI, so to account for the jitter, the wait time is set for x + (z-y). In practice, this delay is generally permissible for control data, while it would not be appropriate for audio data.

The number of simultaneous play event loops is limited only by the memory and CPU of the host machine. Since at every loop the data used is refreshed, they can be manipulated in real time by changing the data stored in the Player or MusicalEnvironment. Which action function or IOI function is referenced in a Player can be changed. The action functions or IOI functions themselves can be modified. Any of the other data in the Players or MusicalEnvironment can be changed. By changing this data, the resulting musical output can be changed. It is in this manner that a livecoded musical performance is realized.

Such manipulation results in many threads and the potential exists for one thread to be writing data which is accessed by another. One problem of real-time multi-threaded systems is guaranteeing the thread safety of data. Haskell provides safe concurrency in the standard libraries of the Glasgow Haskell Compiler (GHC).

5.6 An Example of How Players Work

Here is an example of how Players work, shown in figure 6.

Consider a timpani Player called "A" who has only two jobs. The first job is to hit the timpani. The second job is to wait for a given amount of time, like that written on a score. He hits the timpani, then he waits, then he hits the timpani again and waits, in a loop until he is asked to stop. Now imagine that this Player is joined by another: Player "B". The second Player has only two jobs. The first is to adjust the tuning of the timpani; the second job is the same as that of the first Player. He tunes the timpani and waits, and then tunes it again and waits, repeating like the first Player.

The first timpani Player is a Player stored under the key "A" in the Player store. Its action function is "hit the timpani", which may correspond to triggering a synthdef on scserver called "timpani", which results in a timpani sound being played. The second Player is called "B", and its action function, "tune timpani", is to change the frequency parameter used by the "hit the timpani" function. Each of them has its own IOI function.

Let's expand the situation to include two more Players, Players "C" and "D", who correspond to Players "A" and "B" but are involved with another timpani. The resulting sound is two timpanis being played at the same time. In this case, the "hit the timpani" action is designed to use the name of the Player to determine which frequency should be used. In the same way, the "tune timpani" function uses the Player name to determine which frequency it is tuning and which frequency to tune to.

Now, interestingly, we'll add a fifth Player, who is starting and stopping the Players above. Its action function cycles through a list of actions. Its first action is to start Player "A". Its second action is to start Player "B". Its third action could be to start Players "C" and "D" simultaneuously. Its fourth action could be to pause Players "A", "B", and "D". The design of any action is up to the intentions of the musician.

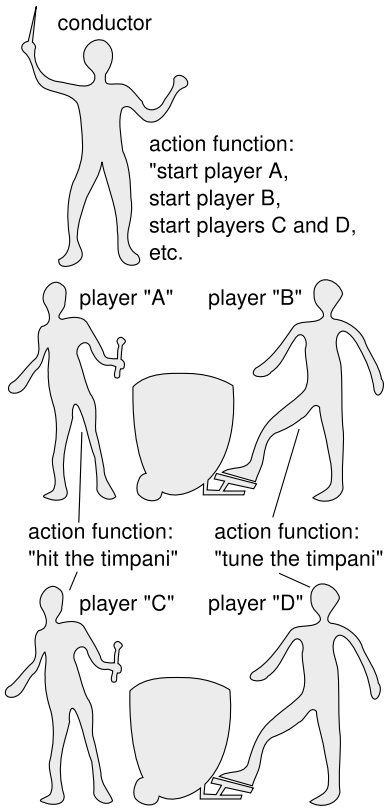

Figure 6: An example of Players at work

5.7 Code Examples of Conductive Usage

A rudimentary sample of usage and corresponding code is given below.

First, the relevant Haskell modules must be

imported, which is accomplished by loading a Haskell document containing the necessary import statements.

```
:load Conductive.hs
```

This example uses SuperCollider, so a convenience command which sets up a group on scserver is called.

```
defaultSCGroup
```

A default MusicalEnvironment is instantiated. It is assigned to the variable "e".

```
e <- defaultMusicalEnvironment
```

An scserver-based sampler is instantiated using this command, which also creates the necessary Players and action functions in the MusicalEnvironment. The function takes a path and the MusicalEnvironment as arguments.

```
s <- initializeSampler "../sounds/*" e
```

All of the Players in a specified MusicalEnvironment can be started with the playAll function. The argument, like above, is the MusicalEnvironment.

```
playAll e
```

The status of all the players in a specified MusicalEnvironment can be viewed with the displayPlayers command.

```
displayPlayers e
```

A list of players can be paused using the pauseN function. The specified players will be looked up in the MusicalEnvironment.

```
pauseN e ["sampler1","sampler2"]
```

Those players could be restarted at a specified time, in this case the first beat of the 16th measure, using the playNAt function. The string after "e" is the specified time, given in terms of measure and beat.

```
playNAt e "15.0" ["sampler1","sampler2"]
```

The tempo of a particular TempoClock can be changed with the changeTempo function. The string "default" is the name of the TempoClock that is to be manipulated.

```
changeTempo e "default" 130
```

A new IOI function can be created. This function call gives the name "newIOI" to an IOI function which will be stored in the MusicalEnvironment. That string is followed by the offset, the number of beats before the first event takes place. The list contains IOI values; in this case, an interval of three beats passes between the first two events.

```
newIOIFunctionAndIOIList e "newIOI"
    0 [3,0.25,1,0.5,2,0.25,3]
```

A player can be told to use this new IOI function by calling the swapIOI function. After specifying the MusicalEnvironment, the name of the player and the name of the IOI function are given.

```
swapIOI e "sampler2" "newIOIPattern"
```

All of the players can be stopped with the stopAll function.

```
stopAll e
```

6 Conclusion and Future Directions

Rudimentary livecoding performances were made possible. The timing was found to be adequate for musical performances, though millisecond timing errors remain. While the library was sufficient for very basic performances, it was necessary to create additional libraries for control and algorithmic composition to achieve a usable interface and more sophisticated performances.

This library alone was far from sufficient to replace current GUI sequencers for most users, though it is hoped this is a good foundation for further research in this direction.

An evaluation method to quantify the usability of this approach should be considered. Additionally, determining the performance of this system versus sclang, Impromptu and others may be valuable.

The library will be tested in performance situations and expanded to be a more complete integrated development environment and performance tool for livecoding performers. Its use in other real-time music applications will also be tested.

The jitter described above is believed to be at least in part due to the garbage collection routines of GHC. Improvements to the GHC

garbage collector are currently being made by its developers. [Marlow, 2010] It is hoped that the gains they make will carry over positively to the performance of this system in terms of reduced delays. There could be other contributing factors, but they have not yet been identified. A deeper investigation into potential causes of jitter and their solutions needs to be undertaken.

Another serious problem involves tempo changes. If the tempo is changed while a play process is sleeping, the next event in that process will be out of sync: early if the tempo is reduced, or late if the tempo is increased. Following events, however, will occur at the correct times. This is because the function for awakening the sleeping Player is unaware of tempo changes and thus cannot adjust the time accordingly. A revised method for sleeping threads which is tempo-aware should be developed.

An important next step is developing a library to make it easy to use MIDI devices with Conductive.

Use of this library by visually-impaired users should be examined, as this text interface may offer such users increased usability. It will be necessary to find users with a braille display and familiarity with vim or emacs for usability testing.

7 Acknowledgements

Thanks are due to Masaaki Tsuji, who provided a great deal of valuable discussion during the development of this library, Naotoshi Osaka for feedback and pushing me ahead, and Michael Chinen, who provided a helpful evaluation. Thanks also to Marvin Wardi for assistance in proofreading.

References

Nick Collins, Alex McLean, Julian Rohrhuber, and Adrian Ward. 2003. Live coding in laptop performance. *Organised Sound*, 8(03):321–330.

C. Coutinho. 2010. tslime.vim. `http://www.vim.org/scripts/script.php?script\_id=3023`, March.

Rohan Drape. 2009. Haskell supercollider, a tutorial. `http://www.slavepianos.org/rd/sw/hsc3/`.

Rohan Drape. 2010. hosc 0.8. `http://hackage.haskell.org/package/hosc-0.8`, March.

H. James Harkins, 2009. *A Practical Guide to Patterns*. `http://www.dewdrop-world.net/sc3/sym09/`.

Nicholas J. Humfrey. 2005. JackMiniMix. `http://www.aelius.com/njh/jackminimix/`, June.

Simon Marlow. 2010. First results from GHC's new garbage collector. `http://hackage.haskell.org/trac/ghc/blog/new-gc-preview`, September.

James Mccartney. 2002. Rethinking the Computer Music Language: SuperCollider. *Comput. Music J.*, 26(4):61–68.

J. McCartney. 2010. SuperCollider Documentation. `http://www.audiosynth.com`.

Alex McLean and Geraint Wiggins. 2010. Tidal - Pattern Language for the Live Coding of Music. In *Proceedings of the 7th Sound and Music Computing conference*.

Alex McLean. 2004. Hacking Perl in nightclubs. `http://www.perl.com/pub/a/2004/08/31/livecode.html`.

Bram Moolenaar. 2011. Vim. `http://www.vim.org/`.

Richard Parncutt. 1994. A Perceptual Model of Pulse Salience and Metrical Accent in Musical Rhythms. *Music Perception: An Interdisciplinary Journal*, 11(4).

Simon L. Peyton Jones, Cordy Hall, Kevin Hammond, Jones Cordy, Hall Kevin, Will Partain, and Phil Wadler. 1992. The Glasgow Haskell compiler: a technical overview.

A. Sorensen and A. R. Brown. 2007. aa-cell in Practice: An approach to musical live coding. In *Proceedings of the International Computer Music Conference*.

Graham Wakefield and Wesley Smith. 2007. Using Lua for Audiovisual Compostion. In *International Computer Music Conference*.

Ge Wang. 2004. On-the-fly Programming: Using Code as an Expressive Musical Instrument. In *In Proceedings of the International Conference on New Interfaces for Musical Expression*, pages 138–143.