Fugue: Musical Programming on the Web

Philip Del Vecchio
University of Pennsylvania,
delphi@seas.upenn.edu

Abstract. Music is an inherently functional domain - it is built on abstractions and functions of notes, themselves abstractions of sound waves. Music is a regular language, i.e., one that can be expressed as a regular expression (sheet music) and recognized by a finite automata (the musician). This project presents a domain-specific language for ClojureScript, a dialect of Lisp that compiles to JavaScript, that allows real-time music composition, production, and performance on the web. By leveraging the Web Audio and Web MIDI APIs, the portability of web applications, and the flexibility and power of Lisp, this project gives programmers and musicians the ability to write and perform music using written code, a command-line interface, MIDI instruments, or real instruments natively from a web browser.

Keywords: Web audio, MIDI, Clojure, Lisp, JavaScript

1 Introduction

Since the dawn of computing, programmers have been interested in the possibilities of computer music. In 1957, Max Mathews wrote MUSIC[1], the first widely used program for sound generation on a computer. In the subsequent half century, dozens of languages, frameworks, and applications for computer music have made computers an essential part of a musician’s toolkit: virtually every part of the music-making process, from composition to synthesis to distribution, now takes place almost exclusively in software. The availability of such software places powerful musical tools, once only available as high-end hardware found exclusively in professional studios, in the hands of amateur musicians and audio engineers.

Despite the ubiquity of musical software, developers of audio applications are often limited to dated, heavyweight, or otherwise cumbersome tools. Languages like C++ and proprietary tools like Steinberg’s Virtual Studio Technology SDK or Apple’s Audio Unit plug-in architecture, though performant, impede the developer with complexity and are not conductive to the experimentation and creativity of music. Programmers and musicians alike rely on high levels of abstraction and feedback to manage complexity and information flow – having to focus on low level implementation details restricts creativity. Programmers of music software seek high-level ways of representing musical structures, instruments, and ideas.
1.1 Web Audio

The web has grown far beyond its original intention to become a platform for lightweight applications on a variety of devices. Multimedia applications have become standard, with website payloads averaging the size of a standard desktop app from a decade ago. To provide web developers with tools for creating audio applications, W3C has introduced the Web Audio API[4]. The Web Audio API is a set of JavaScript functions that allow creation and manipulation of sound in the browser. The API represents the flow of audio internally as a graph, with individual units called AudioNodes that serve as the vertices of the graph.

The API provides tools for the basic building blocks of audio, including oscillators, filters, and tools for volume, pan, and convolution control. While the API provides a moderate level abstraction from the bit-by-bit manipulations required in lower-level languages, it by design provides only the most rudimentary tools for audio, and applications built with the library quickly encounter repetitive boilerplate code and quirks with garbage collection of nodes.

1.2 Web MIDI

MIDI is a standard for digital communications between audio devices that has become a powerful tool for audio engineers and musicians dealing with hardware to control software. MIDI signals are packets of three bytes of data representing the type, key, and value of a control signal or note. Notes are represented by the note-on or note-off type, a key representing the note played, and a value
corresponding to the velocity with which the note was struck. The Web MIDI API, implemented in the stable version of Google Chrome in 2015, gives web applications access (with permission) to signals captured from external MIDI devices using a simple callback. The API provides little functionality beyond capturing input notes and sending output notes.

1.3 ClojureScript

Clojure is a modern dialect of Lisp that compiles to the Java Runtime Machine. It stresses immutable data, has strong concurrency support, and inherits from its parent language the code-as-data philosophy and focus on higher-order functions. The creators of Clojure have also written a Clojure to JavaScript compiler, creating a sublanguage known as “ClojureScript” that brings Clojure to the web.

A musical programming environment consists of a high level language and software tools for interacting with that language that allow audio to be created and manipulated by code written interactively with an interpreter. One existing environment, which served as the inspiration for this project, is Overtone[2], a Clojure library for interacting with SuperCollider[3], a real-time audio programming language. Overtone permits interaction using a real-eval-print-loop, a command line for entering Lisp code that is evaluated immediately.

1.4 Goals

The goal of this project is to bring Clojure audio, REPL-based interactivity, and robust MIDI support to the web. This takes the form of two sub-projects: a ClojureScript library for browser music and a frontend web application for interacting with this library. This frontend web application should be self-hosting in that it should not require communication with a server for compilation of ClojureScript. Musicians benefit from having a lightweight web page that can server as a MIDI instrument host or musical sketchpad, and programmers benefit from having the shortest possible feedback loop from idea to sound.

2 ClojureScript Library

The core of fugue is a ClojureScript library for browser audio and MIDI. It is intended to be included in another project as a dependency to help frontend ClojureScript applications use the musical APIs idiomatically.

2.1 Audio

The Web Audio API presents a model that is fundamentally Object-Oriented. AudioNodes, the building block of the API, are created using methods of a global audio context object. AudioNodes implement the .connect() method to route audio between nodes. AudioNodes connected to the destination of the global
context route audio to the browser’s audio output (usually the user’s speakers). Example AudioNodes include OscillatorNodes, which generate audio based on waveforms, and BiquadFilterNodes, which apply various filters to input audio. The properties of these nodes, such as oscillator frequency and delay time, are controlled by AudioParams, which have associated methods for controlling their values through time, as well as connecting AudioNodes to AudioParams.

ClojureScript, however, is a functional language. To bridge the connection between the paradigms, fugue uses functions that take and return AudioNodes to simulate the flow of audio.

**Fig. 2.** The sin-osc function plays a sine wave.

\[(\text{sin-osc } 440)\]

**Fig. 3.** The lpf function applies a low pass filter to the input.

\[(\text{lpf } (\text{sin-osc } 440) 220)\]

**Fig. 4.** The out function sends output to destination.

\[(\text{out } (\text{saw } 700))\]

**Fig. 5.** The sample function plays audio from a url.

\[(\text{sample } "\text{http://example.com/drums.ogg}"
\]

The arguments that these functions take can be numbers or Clojure channels. Clojure’s core.async library is a CSP implementation that uses channels as a push-pull mechanism. This allows declarative programming about event flow, making for clean concurrent code. The channels passed to audio functions take a map of value and time, where the value is the value to set to and time is the time from the previously scheduled event that the next value should be set. If the time is zero, the value is set immediately and all scheduled values are canceled. This allows for powerful, simple abstractions when asynchronous MIDI events become involved.
2.2 MIDI

The Web MIDI API provides functionality for setting a callback when a MIDI message is received by an external device. Fugue treats MIDI sources as first-class objects, and provides functions for routing the output of these sources:

![Fig. 6. MIDI notes are converted to frequencies for oscillators](sin-osc (:freq (midi-in "Keyboard49")))

In fugue, MIDI, like audio params, uses core.async channels for routing. This allows for creative routing of MIDI into audio devices and direct control using MIDI devices:

![Fig. 7. MIDI 23 controlling a filter frequency from 20Hz to 10kHz](lpf (sample "drum.ogg") (midi-ctrl "Kontrol" 23 20 10000))

Clojure uses the concept of transducers to abstract functions from the data structures they operate on. For example, transducers can be used to apply a function to a list of data or a stream of data, increasing reusability and composable. Fugue uses transducers for “MIDI Effects”: functions applied to an abstract collection of MIDI events to perform transformations. The simplest example is transposing all input notes up or down by a given number of semitones. The same function can be applied to a set of notes or all notes on a MIDI channel.

2.3 Timing

Timing with JavaScript is difficult because the browser is not expected to keep sub-millisecond timing accurately. Scheduling functions with JavaScript can create massive time sways that become an issue when creating metronomes or playing music in time. To rectify this issue, fugue schedules timing functions slightly before they are due, and then compensates for the delay using the Web Audio API’s audio clock, which is useful only for short delays. This solution proves accurate enough for all but the strictest applications.

2.4 Music Theory

Fugue includes a library of music theory functions for composition and experimentation. Chords, scales, arpeggios, time signatures, dynamics, and other fundamentals of Western music are built around functions of pitch and time, which are well modeled by Clojure. A fugue in Western music tradition is a piece based around a theme - the theme is introduced by one voice, then repeated by another,
and another, building into twisted variations on the original theme, inverted, reversed, mirrored, or repeated on itself. The fugue is a functional composition, and this library provides the tools to model and create it.

3 Front end

To take full advantage of fugue requires a read-eval-print-loop, a program that reads input from a prompt, evaluates it, prints the result to the user, and repeats. With a REPL, one can interact with fugue and create music using its various functions. For demonstration, I created a static (offline) web page with embedded JavaScript that evaluates ClojureScript using a version of the ClojureScript compiler written in ClojureScript. This bootstrapped method allows users to directly type ClojureScript commands into the browser and evaluate them without the need to install software on their own machine or communicate with a server.

3.1 Bootstrapped ClojureScript

The developers of Clojure completed a project to port the compiler to ClojureScript in late 2015. Using the `eval` function, compiled ClojureScript code can evaluate ClojureScript expressions. Using the replumb library [5], the fugue app loads the Clojure core library and the fugue library on page load.

3.2 Live Programming

Using the code editor and the REPL, a programmer can compose and perform music with no setup and minimal experience. Sounds created live can be manipulated, saved, and routed arbitrarily. A programmer/musician pair could program and play synthesizers simultaneously, or broadcast their performances over the web.

3.3 Latency

One reason most audio software uses C++ and other low-level languages and frameworks is because of the importance of performance in audio applications: latency introduced by software can make live performance impossible and audio a frustrating listening experience. Modern machines, however, have extremely powerful processors that do not struggle with heavy web applications and multimedia content. Web browsers like Google Chrome have well-optimized JavaScript engines that make high-level audio programming possible. The benchmark I set for latency for fugue was 5 ms under normal load. Below 5 ms, latency is undetectable in all but the most sensitive of systems, and latency did not become an issue in any normal use. Heavy load, such as the creation of thousands of oscillators, can cause the web page to become unresponsive.
4 Discussion

I was able to accomplish most of the original goals for fugue: live audio programming in the browser with plug-and-play MIDI support successfully runs from an offline web browser, and music can be composed and performed from a web browser on a variety of devices. One of the original goals (which is now left as further work) was to allow multiple musicians to program music simultaneously from different machines over the internet. While the framework and design is in place for this, time constraints prevented its inclusion in this project. There is also an interesting potential for educational programming software, as audio is a fun domain to teach programming in.

4.1 Ethics and privacy

Fugue is free software, released under the GPL license. A major goal of fugue was to run locally in the browser, not in “the cloud”, so that users can maintain control over the software they are running. The source code is available publicly at http://github.com/pdv/fugue.

References