FoxDot
Live Coding with Python and SuperCollider

Ryan Kirkbride

University of Leeds, Leeds, United Kingdom
sc10rpk@leeds.ac.uk

Abstract. Live Coding is an electronic musical movement that is growing in popularity as an interface for musical expression where laptop performers program in front audiences; executing, editing, and re-executing blocks of code to generate music. The languages used in Live Coding are usually designed specifically for the purpose of creating music and distance themselves from the traditional paradigms of more general-purpose languages such as Java or Python. FoxDot is an application that bridges this gap to bring the art of performance programming and the science of software engineering together to create music in a way that is accessible to coders or composers; novices and experts alike.

Keywords: Live coding, laptop performance, interactive programming.

Introduction

When I first encountered Live Coding, a method of using programming languages to make music, I had just begun a Master of Arts degree in Computer Music at the University of Leeds. It wasn't long after I had completed my bachelor's in Computer Science and I considered myself a good programmer and, having played music outside of academia for most of my life, a novice in musical composition. However, my first encounter with SuperCollider (McCartney 2002) made me feel like I couldn't do either of these things. Over the course of my master's I was fortunate enough to be taught by Alex McLean, of Live Coding acts Canute (McLean 2015) and Slub (Collins et al. 2003), and starting scouring the web for anything Live Coding related. It didn't take long before I was getting to grips with Tidal (McLean 2014) and SuperCollider, among other languages, but I still couldn't express the musical ideas I wanted to with their capabilities. Furthermore, Live Coding languages tend to be domain-specific (or at least domain-specific implementations of more general-purpose languages) (Guzdial 2014) and structured in a way that didn't fit with the Object-Orientated Programming (OOP) paradigm I had become accustomed to during my undergraduate studies. For example, Tidal is embedded in the language Haskell, which utilises a functional programming paradigm, and the Ruby-based Live Coding language, SonicPi (Aaron and Blackwell 2013), uses a form of procedural programming. OOP is used to represent complex and real-world systems (Kindler and Krivy 2011) and I argue that music making can be as complex as any system found in the real world.

With FoxDot I wanted to create an application that bridged the gap between software engineering and Live Coding so that users who were entry level to programming, composition, or both would still be able to grasp the concepts and make music, while being able to apply the theory to both fields. This article begins by outlining the goals that I wanted to achieve and the purpose for creating FoxDot, and its Interactive Development Editor (IDE), before discussing the implementation of key features and their syntax. It is then concluded with a short discussion about possible future directions for this work.

Design

Before development could begin the technical requirements for the project were outlined; the system should:

- Create music-playing objects that have a state that changes over time (i.e. the note being played) and this state can be accessed by any other object at any point

- Make use of time-dependent variables that, when accessed, return a specific value depending on the time in a metronome
- Utilise a global and dynamic variable system: an object's default values should be accessible via the alteration of a global variable e.g. changing a default scale variable should update all objects using that scale
- Be written in, and derive its syntax from, an existing high-level language that has a large user community and in-depth documentation but also inherit common syntax from other Live Coding languages

Development

Choice of Language

Of the three most popular programming languages in the world (Java, Python, and PHP)[1], Python (http://python.org) is the language that fits the FoxDot requirements best. The rationale for choosing Python for a Live Coding environment is that the combination of its heavy use of OOP and class customisation allows for a flexible design model, and its focus on code readability makes it ideal for use in a performance context.

Alongside Python, SuperCollider will be used to synthesise sounds using Open Sound Control (OSC) (Wright, Freed, and others 1997) in a similar vein to the live coding language "ixilang" (Magnusson 2011). While "ixilang" has a similar implementation design to FoxDot, it is written in the programming language used to write SuperCollider, SCLang, and does not have the established popularity and support that is offered by a language like Python. Another language, SuperDirt (Rohrhuber 2016), is currently being developed that acts as interface between Tidal and SuperCollider, but it is still in an early and experimental phase of development and, as mentioned earlier, does not adhere to the OOP paradigm that is central to the principles of FoxDot.

Interface

FoxDot uses a custom IDE written in its base language, Python, that can execute the 'block' of code (consecutive lines of text with no empty lines) that the text cursor is in by pressing 'Ctrl+Return'. It shares many similarities to the interactive interpreter that comes packaged with the standard installation of Python but allows for the user to easily edit and re-execute code instead of executing each line as it is typed in. The editor also features a console output that displays the Python code executed and any printed output from it, allowing users to program in Python in a much more interactive way than previously available.

Player Objects

In FoxDot, music is performed by creating Player Objects (POs) that take several keyword arguments. Instead of defining a new PO for each sound the user wants to create we define one main class that takes a SuperCollider SynthDef as an argument and sends OSC messages to SuperCollider to create a sound. The first argument is a string that refers to the name of the SuperCollider SynthDef to be used and the second argument is the degree (the index of the note of the scale, which is 0 by default). The duration of each note can also be specified and this value is used in the scheduling process (see TempoClock for more information). Other keywords can be specified that correlate to the keyword arguments used in the specified SynthDef. To create a PO that uses a SynthDef named "pads" that plays the first 8 notes of the default scale using 1/2 beats, the following syntax can be used:

```
p >> pads(range(8), dur=1/2)
p = Player('pads', range(8), dur=1/2)
```

[1] Source: http://pypl.github.io/PYPL.html, Accessed: 13-02-16

These lines are equivalent but the first line has a much cleaner syntax and implies that "pads" is a Python object itself when, in reality, it is not. FoxDot examines each block of code before it is executed and detects when special FoxDot syntax is used (the >> assignment syntax in this case). When creating a PO, the first argument is always the degree(s) of the PO's scale (a globally defined default scale is used unless specified with a keyword argument), which is followed by keyword arguments such as note length ('dur') or sustain ('sus'). Player Objects can play simultaneous notes (useful if you want to play chords, for example) by grouping multiple degree values using a tuple by enclosing values in round brackets '()'. For example:

```
p >> pads([0,2,4,(0,2,4)])
```

This snippet of code creates a new PO that plays the first, third, and fifth note of the scale[2] and then all three of these notes simultaneously. FoxDot automatically laces any nested lists in Player Objects such that the nested list [[0,1,2,3],7] would be equivalent to the list [0,7,1,7,2,7,3,7].

A special PO, known as a SamplePlayer Object (SPO), can be used to play back samples and is created using a $ sign and a string of characters (in a similar syntax to Tidal). The following line of code plays a kick drum ('x'), closed hi-hat ('-'), snare drum ('o'), and another closed hi-hat and repeats:

```
beat $ "x-o-"
```

Each character in the string is mapped to a buffer id used in SuperCollider to play using a SynthDef called "sample_player" and represents one 1/2 beat. A character's duration can be halved by putting them in square brackets. Round brackets are used to lace patterns, similar to nested lists in regular POs, such that the following two lines of code are equivalent:

```
beat $ "x-o-[xx]-o(-[-o])"
beat $ "x-o-[xx]-o-x-o-[xx]-o[-o]"
```

POs are designed to be flexible and accessible. Two POs can be connected by using a PO's "follow()" method, taking another PO as an argument. Attributes, such as the panning or the durations of a PO, are not altered by following a different PO, but when it comes to calculating the note value to send to SuperCollider, note data is retrieved from the source PO and a different frequency is calculated instead. Basic algorithmic composition can be done by using traditional mathematical operators in combination with POs. Adding or subtracting a list of numbers creates a PO expression and modifies its note degree by the value in the list at the index of the current event. A simple way to demonstrate this is with the example below where the PO 'p' will play the same note as 'b' but every third note will be a fifth higher, even when the degree of 'b' is changed.

```
b >> bass([0,2,5,3], dur=1, sus=1/2).stutter(4)
p >> pads(dur=1/4).follow(b) + [0,0,4]
```

TempoClock

A dedicated time-keeping object, known as a TempoClock, is instantiated at runtime that contains an empty queue. POs are added to this queue with a corresponding time value that denotes when they should next be played. The TempoClock plays the PO's next note by calling them as if they were a programmatic function, which means any Python function can also be scheduled to be executed in the future. By default these are scheduled at the start of the next bar as calculated by the TempoClock's time signature attribute.

---

[2] Python, like most programming languages, used zero-based numbering for its arrays whereas musical scales use a one-based numbering system. Consequently, the 1st, 3rd, and 5th notes of a scale are accessed with the indices 0, 2, and 4, respectively.

While running, the clock continually increments its internal counter at the rate of its specified beats-per-minute (BPM). Once this counter is equal to the scheduled time of the first item in its queue, them item is 'popped' from the queue and then called. If the item is a PO, it creates an OSC message based on its current state and sends it to SuperCollider to generate a sound before re-scheduling itself into the TempoClock's queue again. This means notes can be any duration and complex polyrhythms can be created easily by scheduling multiple POs with uneven durations. The code snippet below shows two POs playing the same note but player 'a' plays it three times over the course of two beats and player 'b' plays it four times:

```
a >> pads(dur=2/3)
b >> pads(dur=1/2)
```

Scale Objects

Without the use of the keyword argument "scale" POs use a default scale found in the Scale module (accessed as "Scale.default" from the FoxDot IDE). At start-up it is set to the major scale but can be changed easily (see below). When Scale.default is updated, any Player Object that is using it is also updated (which makes for incredibly easy key changes etc.). The default scale can be changed in a number of ways and can include floating point numbers (as seen in "Scale.justMajor" for example). The following lines of code are equivalent to each other and set the default scale to the natural minor:

```
Scale.default('minor')
Scale.default(Scale.minor)
Scale.default([0,2,3,5,7,8,10])
```

Accessing an element in an array (referred to as lists in Python) is usually done by specifying the index of the item you want to retrieve as a whole integer. FoxDot Scale objects can take floating point values when being accessed in order to return a value between two elements in a Scale. For example; when a Scale object is defined as S=[0,2,4], then S[1.5] will return the midpoint value of the numbers at indices 1 and 2 (in this case, 3). This means that values used for a PO's degree attribute can be floating point numbers and emulate micro-tonal systems quickly and easily.

Time-Dependent Variables

A time dependent variable (it will be referred to as a 'Var' from here on in) is a variable that, when accessed by a PO or user, returns a value derived from the current state of the global TempoClock. These variables are created using the following syntax:

```
a = Var([0,3,4],[8,4,4])
```

The Var above, 'a', returns the values 0 for 8 beats, then 3 for 4 beats, and finally 4 for 4 beats before starting over. This can be very useful for creating multiple Player Objects that share the same underlying music based on chord sequences, like so:

```
a = Var([0,4,5,3],8)
p >> pads(a, dur=1).offbeat() + (0,2,4)
b >> bass(a, dur=1/4, sus=1)
```

Calling the update method on a Var will change the values for all POs that are accessing it, which means patterns can be shared very easily between POs. Vars can be used for any keyword argument, e.g. setting a PO's amplitude to be loud for 8 beats and then silent for 24, and used in PO expressions. Figure 1 is the passage of music equivalent to what would be produced on repeat with three simple lines of code such that both players are following the common chord sequence I V VI IV with 'p' oscillating around the fifth and third notes of the chord, and 'v' playing sustained notes with alternating harmonies.

```
a = Var([0,4,5,3],4)
p >> pads(a + var([4,[2,0],1] [2,1,1]), dur=[1/2,1/4,1/4]) + [0,[-1,1]]
v >> viola(a, dur=4) + (0,var([[9,2],4],4))
```

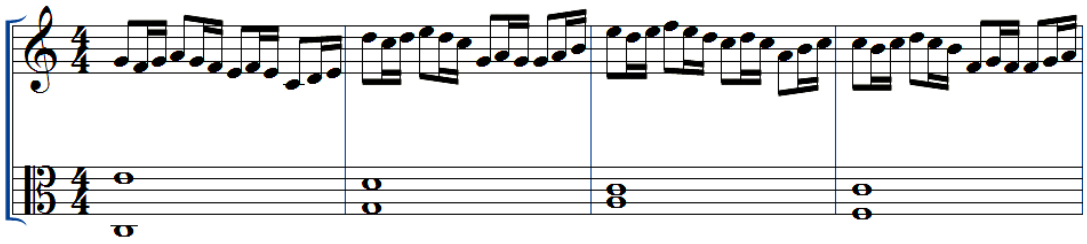

Figure 1: Nominal score representation of output from FoxDot code

When Statements

A common component of any programming language are conditional statements (commonly known as “if” statements), which execute a block of code only if a certain condition is satisfied. FoxDot implements a “when” statement that evaluates this condition at regular points in the TempoClock’s cycle and executes any code assigned to that condition when it is evaluated to be true. Some useful applications of this feature include allowing POs to change their state based on the state of other POs over time, and changing multiple POs’ states based on a variable that may, or may not, be affected by POs. Below is an example of an implementation of two ’parts’ of a piece of music that can be alternated by changing the value of the variable ’val’ to 0 or 1:

```
val = 0
a = Var()
when val == 0:
    a.update([0,-3,-4],[4,4,8])
    p >> pads([0,2,4], dur=[1/2,1/4,1/4])
    b >> bass(a, dur=[1.5,0.5,2], sus=0.5) + Var([0,2],4)
else:
    a.update([7,4,5,3],4)
    p >> pads(a, dur=[1.5,1.5,1], sus=2) + var([2,4],4)
    b >> bass(a, dur=1/4, sus=1) + [0,0,(0,9)]
```

Further Work

Expanding Content

The basic concepts and functionality for FoxDot have already been programmed but there is still a lot of work to be done. The library of samples is very small and needs to be expanded to incorporate more “interesting” sounds in addition to those of a basic drum kit. FoxDot currently comes with a SuperCollider .scd file that contains several SynthDefs already written but, like the sample library, needs to be expanded. FoxDot is designed to be flexible and customisable; both the collections of samples and SynthDefs should be accessible by the user and changeable at their choosing through the use of an easily edited configuration file. Similarly, the types of musical patterns available to the user will also be expanded upon, drawing inspiration from existing definitions (Spiegel 1981).

User Testing

FoxDot is currently in its alpha stage of development and it will be quite some time before a stable release version exists, but between now and that time the collection of user feedback would be useful for adding ideas to, and

furthering the development of, FoxDot. If you would like try FoxDot for yourself, the most up-to-date version is available at https://github.com/Qirky/FoxDot and any feedback will be more than welcome.

References

Aaron, Samuel, and Alan F Blackwell. 2013. "From Sonic Pi to Overtone: Creative Musical Experiences with Domain-Specific and Functional Languages." In *Proceedings of the First ACM SIGPLAN Workshop on Functional Art, Music, Modeling & Design*, 35–46. ACM.

Collins, Nick, Alex McLean, Julian Rohrhuber, and Adrian Ward. 2003. "Live Coding in Laptop Performance." *Organised Sound* 8 (03). Cambridge University Press: 321–30.

Guzdial, Mark. 2014. "Live Coding, Computer Science, and Education." *Collaboration and Learning Through Live Coding (Dagstuhl Seminar 13382)* 3 (9). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik: 162.

Kindler, Eugene, and Ivan Krivy. 2011. "Object-Oriented Simulation of Systems with Sophisticated Control." *International Journal of General Systems* 40 (3). Taylor & Francis: 313–43.

Magnusson, Thor. 2011. "Ixi Lang: A SuperCollider Parasite for Live Coding." In *Proceedings of International Computer Music Conference*, 503–6. University of Huddersfield.

McCartney, James. 2002. "Rethinking the Computer Music Language: SuperCollider." *Computer Music Journal* 26 (4). MIT Press: 61–68.

McLean, Alex. 2014. "Making Programming Languages to Dance to: Live Coding with Tidal." In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Functional Art, Music, Modelling & Design*, 63–70. ACM.

McLean, Alex. 2014. 2015. "Reflections on Live Coding Collaboration." In *XCoAx 2015: Proceedings of the Third Conference on Computation, Communication, Aesthetics and X*, 213.

Rohrhuber, Julian. 2016. https://github.com/musikinformatik/SuperDirt.

Spiegel, Laurie. 1981. "Manipulations of Musical Patterns." In *Proceedings of the Symposium on Small Computers and the Arts*, 19–22.

Wright, Matthew, Adrian Freed, and others. 1997. "Open Sound Control: A New Protocol for Communicating with Sound Synthesizers." In *Proceedings of the 1997 International Computer Music Conference*, 10. 8. International Computer Music Association San Francisco.