Programming in Time: New Implications for Temporality in Live Coding

Ryan Kirkbride
Interdisciplinary Centre for Scientific Research in Music (ICSRiM), University of Leeds
sc10rpk@leeds.ac.uk

ABSTRACT

Traditional western music is a temporally-dynamic system built on inter-dependent and time-varying relationships between rhythms and pitches. This paper provides an insight into a means of abstraction for representing such a system in Live Coding through the use of time-varying values that interact with one another to create a template for organic musical performances. As producing this behaviour in computer programming languages can prove difficult, this paper will also discuss the problems that temporality can present in Live Coding and the methods used to address them. Finally, the novel programmatic function, “time-dependent variables”, is presented as part of the Python-based Live Coding system, FoxDot.

1 Introduction

One possible definition for Live Coding could be that it is a form of artistic expression that is mediated through the use of programming languages and exists within the temporal domain. Without consideration of the nature of time, Live Coding would not be live at all and its artistic merit would only be derived from how well a programmer follows best practices. This paper presents the notion that programming in time is an aspect of Live Coding that should be celebrated, and demonstrates new and existing methods for both utilising and circumventing its temporal nature. Firstly, I will discuss the aspects of time-keeping that have posed problems for Live Coders in the past and the methods used overcome them, which provides a segue into the introduction of the Python-based Live Coding language, FoxDot, and its temporal functionalities such as “time-dependent variables” and “when conditions”.

2 Temporality in Live Coding

Temporality is the nexus of many types of music-making but its existence raises the question; how should time be represented in music? We usually think of time in music as the rhythms and phrases created using syntax from traditional western music notation but it has been clear throughout the last half-century that composers have continually challenged this concept. An advantage of using a programming language to make music is that not only can you represent time, but you can also exploit it. One example of this exploitation can be seen in the Just-In-Time Library (JITLib), developed by Julian Rohrhuber (Collins et al. 2003), that allows the programmer to refer to place-holder “proxy” variables before they have been assigned a value; a far cry from the temporal order of program execution in standard programming procedure.

Circularity is another implementation of time in Live Coded music and this is utilised in several different environments, such as Sonic Pi (Aaron, Orchard, and Blackwell 2014) and TidalCycles (McLean and Wiggins 2010), but the means of doing so can vary from language to language. SonicPi implements traditional programming constructs, such as loops, to cycle over musical events whereas TidalCycles uses functions of time to create musical events from cyclical pattern structures. An internal circularity can also be implemented to schedule musical events, which is done through the use of temporal recursion; functions that execute then call themselves to be executed in the future. Recent examples of this temporal construct can be found in the programming environments Impromptu (Brown and Sorensen 2007; A. Sorensen and Gardner 2010) and Extempore (Swift et al. 2013) but the concept itself dates back to 1996 (Sorensen 2013) in early versions of the popular sound-synthesis language, SuperCollider (McCartney 2002). The examples discussed here are just some of the techniques employed by Live Coders that demonstrate how programming languages can circumvent the temporal obstacles presented in laptop performance but have played an important role in the development of the Python-based Live Coding environment, FoxDot.

1One such example of this is John Cage’s Musicircus (1967)
2https://github.com/Qirky/FoxDot
2.1 Live Coding with Python and SuperCollider

FoxDot is a Python-based musical Live Coding environment that utilises an object-oriented programming (OOP) paradigm. OOP is useful for representing complex and real-world systems (Kindler and Krivy 2011) and I argue that the conceptual nature of creating music is as complex as any system found in the real world. The rationale for choosing Python (Rossum 1996) as the host language for a Live Coding environment was that the combination of its large use of OOP and class customisation would allow for a flexible design model, and its focus on code readability would make it ideal for use in a performance setting. Python is one of the most popular programming languages in the world (Cass 2015) and adapting it for use in laptop performance makes Live Coding accessible to a large number of existing programmers who already know its syntax and semantics. Over the years programming languages have changed in many ways and I will now demonstrate both why, and how, FoxDot is well-adapted to dealing with the problems in trying to represent the dynamic and temporal-centric system of music.

3 Time Dependent Variables

3.1 Motivation

In traditional western music nominal relationships exist between pitches and rhythms that change continuously over time. Replicating this time-varying behaviour in programming languages, however, is not always a simple task. In the Python example below, the variable X is assigned a value of 2 and Y is assigned the value of X plus 2, which is 4. Typically, when X is then increased by 3, the value of Y does not change because there is no reference to it stored in X. However, in “reactive” programming (Bainomugisha et al. 2013) any changes in a variable’s content are propagated to its dependencies as defined by a data flow such that, in our example, the value of Y would also be increased by 3 when X was updated.

```python
>>> X = 2
>>> Y = X + 2
>>> X = X + 3
>>> print X, Y
5 4
```

Examining the example from a musical perspective, Y might correspond to a note that is being played a third above the note X. When X changes, the desired response would be a change in Y relative to the change in X such that it is still a third above X. Not only should the relationships between musical elements react to dynamic events, but they should change and evolve over time. This type of behaviour is quite difficult to produce in a typically static programming language such as Python, but the addition of reactive and time-dependent variables has made this possible in FoxDot.

3.2 Implementation

3.2.1 Functionality

In FoxDot time is managed using a designated time-keeper called Clock, similar to the abstraction of SuperCollider’s “TempoClock” used by Bell (2011). Clock is used in the scheduling of musical events using an internal counter that increases over time with respect to its bpm value. A time-dependent variable (TDV) is created by instantiating a var object with two arguments. The first is a list of values, or states, that represent the different possible states of the variable (stored in the object’s data attribute). The second is a list of durations, in beats, that the variable should spend in each of these states, although a single value can be used if the TDV is to spend an equal amount of time in each state. The reason these arguments are separate and not given as a single list of value-duration pairs is so that they can be of unequal lengths and can incorporate list-generating functions such as range().

The segment of FoxDot code below shows the code used to create a variable that contains a value of 0 for 6 clock beats then a value of 1 for the next 2 clock beats. At the end of this time period, a total of 8 beats, the value is set back to 0 and the cycle repeats.

```python
>>> x = var([0,1],[6,2])
>>> print x # t = 0
0
>>> print x # t = 7
```

The current state of a TDV is calculated whenever the `var.now()` method (code below), a function of time similar to those found in TidalCycles, is called. This is done when another function, such as printing to screen, wants to access its contents.

```python
>>> print x # t = 10
0
```

The current state of a TDV is calculated whenever the `var.now()` method (code below), a function of time similar to those found in TidalCycles, is called. This is done when another function, such as printing to screen, wants to access its contents.

```python
class var:
    ...
    def now(self):
        # self.metro.now() -> reference to number of the Clock's beats
        t = self.metro.now() % self.length()
        # Iterate over the possible values
        for i in range(len(self.data)):
            value = self.data[i]
            # If the clock counter falls within these start and end times, break out of the loop
            if self.time[i][0] <= t <= self.time[i][1]:
                break
        return self.calculate(value)
```

The `var.now()` code above shows how modulo division of the Clock counter is used to calculate the time relative to the start of a `var` instance and return the object’s current state. You may notice that a method, `calculate`, is called on this value before it is returned; this checks if the current `var` instance has any dependencies and is updated accordingly.

```python
>>> a = var([0,1],2)
>>> b = a + 1
>>> c = b * 2
>>> print a, b, c # t = 0
0 1 2
>>> print a, b, c # t = 2
1 2 4
>>> a = var([5,10], 2)
>>> print a, b, c # t = 4
5 6 12
>>> print a, b, c # t = 6
10 11 22
```

In some object-oriented programming languages, such as Python, there is an added layer of rules on top of type-checking called “duck typing”. Where type-checking is concerned with making sure variables are the appropriate data-type when being used, duck typing is concerned with making sure the program constructs behave in an appropriate way (Martelli 2000)3. Thanks to Python’s duck-typing philosophy, user-defined classes such as `var` can emulate properties of standard data types such as integers, strings, and even container types like lists (commonly known as arrays). They can do so by using what are commonly referred to as “magic methods” (Kettler 2012). An example of an implementation of one of these “magic methods” would be overriding the `__str__()` method of an object and forcing the `var.now()` method to be called so that string representation of the instance’s `state` is returned whenever a print function is called on it. All of an object’s mathematical operator behaviours, such as addition or multiplication, can be manipulated in a similar way. It is possible to override a `var` object’s `__add__()` method to decide what happens when a number is added to it, like in the first two lines in the FoxDot code above. The line `b = a + 1` returns a new `var` instance, equivalent to `var([5], [2])`, but also contains a reference to the original. This way, whenever the new instance’s `now()` method is called to get its current state, it checks the original instance’s state at that moment in time using its `calculate()` method (a sort of pseudo-lazy-evaluation). This works for multiple `var` instances that have been “chained” together like in the example above. This can be used to used to define (and re-define) relationships between pitches and rhythms to create organic sounding music in a fast and effective way. It is also possible for TDVs to use a combination of different data types such as strings and integers and floating point durations (see below).

---

3The name “duck typing” is said to refer to the tongue-in-cheek form of abductive reasoning known as the “duck test”. For more information see the Wikipedia article on the topic.
Sometimes this time-varying behaviour may be unwanted and can be negated if need be. By calling one of Python’s built-in special functions, such as `str()`, `int()`, or `float()`, it possible to statically assign the contents of a TDV to a new variable with the desired type (provided they are compatible), as shown below.

>>> a = var([1, 'one'], [2.5, 1.5])
>>> b = a * 2
>>> print a, b  # t = 0
1 2
>>> print a, b  # t = 2.5
one one

### 3.3 Applications

FoxDot creates music using Player Objects that schedule OSC messages to be sent to SuperCollider based on their current state, as defined by certain musical attributes such as pitch, duration, and sustain. Using a TDV to define pitches shared between multiple Player Objects means that a chord progression can be created and, consequently, manipulated very easily. Even if a Player Object is using a random list of durations the pitch will always be the same as any other Player Object also using the same TDV. TDVs can be used in a range of musical contexts in FoxDot, such as constructing scales, setting the tempo, and defining a Player Object’s duration. This allows for changes to happen over the course of a performance outside of the programmer’s direct control; creating a constantly shifting soundscape.

### 4 Conclusions and Further Work

In summary, this paper describes some of the techniques used by Live Coders to overcome the problems that can occur while programming music with regards to temporality, and outlines a novel method of exploiting this facet of Live Coding to create dynamic musical systems through the use of time-dependent variables. It also introduces the Python-based Live Coding system, FoxDot, that implements this construct in tandem with a reactive and object-oriented programming paradigm. A development of the TDV that is currently a work-in-progress is a subclass of `var` that uses linear interpolation to calculate continual change between a TDV’s values.

Another temporal mechanic that is in development for the FoxDot system is called the “when statement”. Similar to the traditional programming concept of conditional statements (sometimes known as “if statements”), “when statements” execute code once a condition is met. This condition is evaluated at regular time intervals such that a block of code is executed whenever a condition is evaluated as true in order to respond to constant changes that are present in a dynamic system. Below is a trivial example of how this might be used with TDVs to print messages to screen depending on its contents:

```python
>>> number = var([0, 2, 3, 4], [4])
>>> switch = False
>>> when number < 3:
>>>     if switch is False:
>>>         print "Hello, ",
>>>         switch = True
>>>     else:
>>>         if switch is True:
>>>             print "World!"
>>>         switch = False
```

I implore those who are involved with this style of algorithmic music-making to celebrate the temporal aspect of live coding and to continue exploring ways to apply and exploit it.

---

4A full discussion of Player Objects is beyond the scope of this paper, please see the project’s GitHub readme for more information.
References


Martelli, Alex. 2000. https://groups.google.com/forum/?hl=en#!msg/comp.lang.python/CCs2oJdyuZC/NYjla5HKMOIJ.


