Livecoding's many meanings

Andrew Sorensen

Institute for Future Environments

Queensland University of Technology

a.sorensen@qut.edu.au

Ben Swift

Research School of Computer Science

Australian National University

ben.swift@anu.edu.au

Alistair Riddell

School of Art

Australian National University

alisair.riddell@anu.edu.au

Introduction

Musical meaning is predicated on communication, but communication does not entail meaning. Ultimately for any communication of musical meaning to take place between a composer and an audience, some shared interpretation is required, as noted by Boretz:

> Thus the salient characteristic of an art entity may, most generally, be considered to be its "coherence"; and the *extent* of its coherence, and hence of its particularity as a work of art, may be considered to reside in the degree of determinate complexity exhibited in the ordered structure of subentities of which it is a resultant. (Boretz 1970, p.543)

Livecoding (Collins et al. 2003; Wang and Cook 2004) is a performance practice in which meaning exists on a number of different levels. Firstly, meaning is inherent in the formal system that defines the programming language interface used for livecoding. This meaning is known as the *program-process* semantics (Smith 1996). Secondly, meaning is conveyed through the runtime computational processes set in action by the livecoder's code manipulations. This is the *process-task* semantics (Smith 1996). Finally, livecoding's cyberphysical relationship with the physical environment results in perturbations in the world (Sorensen and Gardner 2010). These perturbations result in *embodied* meaning.

Meaning in a livecoding context is therefore multifaceted—a complex interplay of symbolic, computational and embodied meaning. Further complicating these relationships is the fact that they are shared between a livecoding practitioner and an audience, each of whose relationship with livecoding's "meanings" will be unique.

In this essay we attempt to unpack some of livecoding's many meanings, paying particular attention to the formal semantics which are so prominent in livecoding practice, with its commitment to the display of source code and the importance of algorithms. We approach this question largely from a compositional perspective by investigating the structural function of form. We acknowledge that ideas of meaning in music (and indeed in the arts more generally) have been widely discussed elsewhere (Meyer 1956; Goodman 1976; Boretz 1970; Cross and Tolbert 2009). However, we also believe that livecoding offers a fresh challenge to the interrelationships of meaning in formal systems and musical composition. To explore these ideas we expand

on the work of our colleagues (Rohrhuber et al. 2007; Rohrhuber and de Campo 2009; Magnusson 2011a; McLean and Wiggins 2010) in the hope of encouraging further discussion of livecoding's many meanings.

Musical Formalism

A core concern of the musical composer is managing complexity. Musical form, at all levels, requires a delicate balance of coherence and novelty. In order to tame this musical complexity, composers have often turned to formal methods. The desire to impose order on the musical chaos of the times has found voice from antiquity through to the present day (Essl 2007; Edwards 2011; Loy 1989).

> at this time when music has become almost arbitrary and composers refuse to be bound by any rules and principles, detesting the very name of school and law like death itself. (Johann Joseph Fux (writing in 1725), quoted in Fux and Mann 1965, p.20)

Order, or coherence, in music is multifaceted; and one important distinction in this regard is the distinction between *structural* and *cultural* coherence. That structural coherence in music would be amenable to formal processes is largely self evident, form and structure being almost synonyms from a compositional perspective. Cultural coherence, on the other hand, appears to be considerably more difficult to formalise, and is perhaps best tackled

> with that particular kind of exploration that systematically extends perception; a kind of exploration called 'play' (David Keane, quoted in Emmerson 1986, p.111).

Livecoding meets both structural and cultural criteria by supporting structural development through formal methods, and at the same time supporting the systematic extension of perception through play. For the livecoder, the digital computer supports the construction of sonic micro-worlds—creative spaces that support an unprecedented spectrum of sonic possibilities. To fully realise the power of these most flexible of machines the livecoder must work within the framework of formal systems.

Formal systems often play a functional role in musical composition and performance as accompanists, antagonists, muses and even conductors, but are usually heavily directed by a human performer (or performers) working through some form of non-formal interface—a keyboard, joystick, monome, microphone, or similar.

In livecoding the performance is also heavily directed by a human performer (the livecoder), but in this case the interface is *itself* a formal system. What distinguishes livecoding from other formal approaches to music is that the formal system under consideration can be modified on-the-fly by a human operator. In most traditional formal systems contexts (e.g. GenJam Biles (2007)) the rules and axioms are unalterable, once the system is defined it cannot be altered in playback. Livecoding breaks from this rigidity by supporting a human composer who operates "above-the-loop", in that livecoding allows for the run-time modification of the system's axioms and rules. By incorporating a formal language interface into a real-time system composed of both sensors and actuators, livecoding enables composers to modify automatic formal systems designed for music production in real-time at run-time.

Token Meaning and Embodied Meaning

Although many computer music composers are comfortable with formal languages (particularly computer programming languages) the relationship between these formal systems and the musical abstractions which are built upon them are complex and often

unclear.

Haugeland (1981) describes the computer's central processing unit (CPU) as an automatic formal system (AFS) which inputs, stores, manipulates and outputs meaningless tokens. The assertion here being that these tokens are non-symbolic, that they lack referents, in a Fregian sense (Eco 1979). An algorithm is a formal system because it is defined in terms of the form of the representation without regard to any external reference.

This is not to suggest that these meaningless tokens are meaningless internally. Within the formal system tokens must have a consistent semantics in order to support interpretation by the AFS. In other words, they must have a syntactical meaning. What makes an AFS such a powerful tool is that these internal semantic engines can be layered on top of one another, operating at increasingly higher levels of abstraction. This allows programmers to create new conceptual worlds that obey laws that are independent of the platforms on which they are built.

That these conceptual worlds exist presupposes that semantics can operate at many different levels. An important question then is how meaning crosses semantic boarders. Morris (1938) proposed that semiotics be broken into three fields; pragmatics, the relationship between signs and interpreters; semantics, the relation of signs to objects; syntactics, the relations of signs to one another. Zemanek (1966) suggested that in relation to semantic *levels* these fields could be roughly broken down as follows:

> syntactics becomes relations within one level, whatever the level is; semantics becomes relationships between two adjacent levels; and pragmatics presumably becomes the relations leading outside of the level scheme, whatever "outside" is. (Zemanek 1966, p.140)

Our text rich programming languages leverage *linguistic* natural languages. It is

therefore understandable that tokens commonly used in programming languages would have strong symbolic denotations. However, the degree to which those denotations are syntactic, semantic, pragmatic, or a mixture of all three is often vague. A `frog` variable is a syntactic element in the C programming language, and conveys meaning for the interpreter (compiler) of the C language. The `frog` variable also conveys semantic meaning for the human programmer, denoting a frog - being a *sense*, *concept* or *type* of a frog. That these two meanings coexist is interesting within a computing context because the strong "cultural unit" (Eco 1979) that helps to form the frog concept, is only valid so long as a pragmatic relationship with the concept remains valid. In other words, if the programmer uses `frog` as the variable name for an animated character, and a robot is drawn on the screen, the linguistic identity of the symbol 'f-r-o-g' is challenged, and a new sign production takes place.

The indexicality of this sign relation, between the `frog` and a computationally driven robot animation points to a powerful attribute of livecoding - its inherently cyberphysical context.

Consider the token `random()`. Within the context (interpretation) of sound, meaning can be ascribed to this symbol, forming a mental conception of white noise. The context of the interpretation being critical so as not to produce any number of mental images related to `random()` but not to white noise - chaos theory perhaps. Of significance for livecoding, this mapping can be physical as well as conceptual. The symbol `random()` in the context of livecoding can reify the *concept* of white noise into a physical manifestation of white noise sounding in the environment. In other words, livecoding can make sign production an embodied experience, giving `random()` a real-time indexical relationship to white noise in the physical environment.

In the following we start to unpack some of these ideas in a more practical context, starting with the very simple ixi lang (Magnusson 2011b) example in figure 1.

```
drums -> |k   s   k   s   |
```

Figure 1. A simple ixi lang code example.

```
gnal ixi }# epdddbdddpdddbddde
```

Figure 2. A simple gnal ixi code example.

The tokens in this example (`drums`, `->`, `|`, `k` and `s`) are all valid symbols in the AFS specified by ixi lang's creator, Thor Magnusson. These tokens have grammatical but not lexical meaning to ixi lang; there is no need for the kick and snare drum samples to be represented by the symbols `k` and `s`, they could just as easily be `j` and `z`. Grammatically the meaning would remain the same. However, semantically, given the context of this paper, the symbol `drums` and the use of ixi lang it seems reasonable to ascribe the concepts of kick to `k` and snare to `s`. The kick `k` and snare `s` symbols convey meaning in the "conceptual musical world" of rhythm, timbre and musical structure (repetition) which can be understood by *looking at* the token string alone.

Ultimately, the composer has a very human ability to attribute musical meaning to the symbols `k` and `s` as a kick drum and a snare drum, space as silence, `|` as loop boundaries, and position as an indication of temporal structure. This assignment of meaning exists outside of the formal semantics of the ixi lang system. There is a distinction between the (human) musical meaning of a set of tokens and the meaning of those tokens within the formal system in which they are a well-formed string. These different types of meaning are independent, as an example from gnal ixi, ixi lang's bizarro-world cousin, demonstrates. We doubt that anyone would grasp the musical meaning of the tokens in figure 2 without reference to the previous (figure 1) example:

And yet, upon *executing* the expression in figure 2 in gnal ixi, the livecoder will immediately hear the musical result as a repeating kick-snare-kick-snare four-beat pattern. What this demonstrates is the difference between a static statement of formal

```
drums -> |s   k   s   k   |
```

Figure 3. Another ixi lang example, with the symbols changed from figure 1.

symbolic meaning and the idea of a statement "being meaningful". This is significant as it suggests that livecoding's "liveness" provides an active, dynamic and potentially physical "meaning" that is otherwise missing from a static symbolic interpretation of the system. Embodied meaning.

These two examples (figures 1 and 1) demonstrate that "meaning" can happen through a semantic interpretation of tokens by an *human* interpreter, or through the mechanical transduction of formal tokens into the physical environment. In the first example, musical information is conveyed via tokens that are interpreted directly by both the livecoder and the audience. In the second example, musical information is conveyed through a hierarchical nesting of abstraction layers—tokens are interpreted by ixi lang, turned into different tokens which are interpreted by SuperCollider (which ixi lang uses for audio signal processing), turned into different (signal-level) tokens to be interpreted by a DAC, transduced from electrical energy into magnetic energy and pushed out into the world as pressure waves. The difference here is analogous to the difference between reading a score and listening to an orchestra.

Both of these approaches convey meaning, but not the same meaning. A semantic meaning can be formally correct, and yet represent an ambiguous, or false, relationship to the world.

The variation in figure 3 also results in a kick-snare-kick-snare sonic result, however, in this example `s` signifies kick and `k` signifies snare. This obvious but important problem is described by Eco (1979) as the "referential fallacy". A formal system can have a valid semiotic function, and yet be false in the real world. This is of course true in livecoding systems, as with other formal-systems. However, livecoding, with its

```
sine ->  /0.000   0.707   1.000   0.707   0.000   -0.707   -1.000   -0.707/
```

Figure 4. Literal numeric values representing a (very lo-fi) sine wave.

real-time relationship with the physical environment, can support the livecoder in more readily resolving ambiguities between a multiplicity of sign systems. This is a useful consequence of the "liveness" of livecoding.

Formal Structures and Musical Hierarchies

The appeal of a strong musical semantics in the token system is obvious. Musicians who are unfamiliar with programming languages can intuitively grasp what the tokens in the source code mean, and also what changes to the source would be necessary to achieve desired changes to the musical output. However, there is an inherent cost to building these higher-level "conceptual musical worlds"—that a generality inherent in the manipulation of "meaningless" symbols gives way to a structured hierarchy imbued with meaning. Consider a contrived ixi lang sine wave designed as a signal-rate operation, as shown in figure 4:

It is clear that specifying waveforms in this direct pattern-language formalism is unwieldy, perhaps impossibly so. From a formal systems perspective the tokens in this IXI example must be literal values, although what these tokens mean *musically* can differ between different modes of the language—the example above would be a *signal* mode, in addition to the provided *melodic*, *percussive* and *concrete* modes currently supported by ixi lang. This privileges token semantics which are information rich from a musical perspective (such as pitch numbers or sample names) rather than tokens which are musically information poor (such as the raw audio samples offered for interpolation in this example).

It is worth noting that with an appropriate *hidden* interpolation layer, this example

```
(bind-func dsp:DSP
  (lambda (in time channel data)
     (cos (* 2.0 3.141592 440.0 (/ time 44100.0)))))
```

Figure 5. A function which uses the `cos` function to generate a pure sine tone as audio output.

may actually get quite close to the desired (sinewave) result—an IXI pattern language for signals. Nevertheless, there are certainly more economical representations that describe the real-world phenomenon of sound more generally.

It is worth considering that where computer science gains leverage through formal abstraction, engineering gains intellectual leverage through mathematical modelling. This allows engineers to tame an unruly reality, but it does not provide the explicit *interface* or *conceptual world* that computer-science abstractions provide. In other words, the purity of mathematics, its *unintentional* stance (Dennett 1989), divorces it from the type of semantic entailment that higher level computational abstractions may invoke. The value then of an unintentional stance is to lessen (although never to remove) the chance of being caught up in a referential fallacy.

Let us briefly consider the implications of this for formal systems in the domain of sound and music. At the signal level, computer hardware peripherals operate with numbers—for our present discussion we will assume floating point numbers. In the Extempore code example in figure 5, the `dsp` function is called directly from the DAC to "compute" a real-time waveform on a sample-by-sample basis. We anticipate that most readers will have an intuition about the musical structure of this code example—a 440hz (concert A pitch) sine tone. Now consider the code example in 6:

Musically, the example in figure 6 represents the same kick-snare-kick-snare pattern as the ixi lang example in figure 1, with the sinusoid oscillating between the general MIDI "drum" numbers 36 (kick) and 38 (snare). What is interesting about this simple example is the degree to which an intentional representation (that of a signal-level

```
(bind-func drums
  (lambda ()
     (play drums
           (+ 37 (cos (* 2.0 3.141592 0.1 (/ time 44100.0))))
           80
           .1)
     (sys:sleep 5000)))
```

Figure 6. Another use of the `cos` function, but this time in a sequencing role.

sinusoid) almost forces itself upon those who already posses an appropriate system of interpretation. The two examples above help to demonstrate that it is not domain knowledge of the mathematical cosine function, nor of Extempore's XTLang (an introduction to XTLang can be found at http://extempore.moso.com.au) programming language, instead it is a common domain understanding of the cosine's usage in signal processing that gives the first example a clearer musical meaning than the comparatively unusual usage of a cosine for flip-flopping between two MIDI values. From the formal position of syntactically-valid XTLang token strings, there is virtually no difference between the two.

These simple examples have shown the (at times complicated) relationship between the different types of meaning the composer is dealing with in using formal systems for musical expression. One surprising point is the fact that although a musical legibility (a lexical semantics) in token strings offers some benefits, in livecoding this relationship is less necessary because musical meaning can be derived through the algorithms execution and realisation in the world. Having the livecoder present in the loop connects the token system to its embodied musical result, and allows for fuller reflection on the current state of the musical system. This allows the token system itself to be less strongly coupled to the musical domain it seeks to represent, which provides other benefits to the livecoder, as we shall attempt to articulate in the next section.

Meaningless Tokens are Powerful Tokens

We take the idea of a sound object (by which we mean the commonsense definition as the "lowest-level component" or "fundamental building block" of a musical composition) as a good starting point for a musical exploration of the semantic issues discussed in the previous section. The idea of an atomic sound object, made up of a fixed number of discrete, and highly quantized parameters, is largely redundant to the modern computational composer. Instead, the sound object is unstable and composable, and this shifting identity is now a central part of computer music practice. The livecoder is free to choose which attributes define the sound object, what their capacities are, and whether these attributes are stable or unstable over time.

In figure 7, we deliberately conflate elements of what would usually be considered to belong to "discrete event" vs "signal level" abstractions. As in the proceeding `dsp` example (figure 5), we take the DAC's floating point representation as our symbol floor. We include comments in this example for the reader's benefit, although these would not usually be present in a real livecoding context.

This brief Extempore example (figure 7) shows a complete, run-time compiled and on-the-fly modifiable, 'waveform generator'. A small, self contained, musical piece. It includes pitch, dynamic, timing and spectral dimensions. As with the earlier `dsp` function, `caprice` is called on a sample-by-sample basis in order to directly calculate a waveform.

The caprice plays "notes" of stable pitch and constant volume at a rate of 4Hz. It does this not through any built-in concept of a note, but by performing a modulo check on the raw `time` index, a raw counter which increments once per audio sample (at 44.1kHz). If this check returns zero, the code (non-deterministically) changes the values of the local state variables `pitch` and `volume`. In the latter part of the function, these

```
(bind-func caprice 10000
  ;; initialize/allocate delay line and declare local vars
  (let ((dline:|1024,double|* (alloc))
        (pitch 1024.0)
        (volume 0.0)
        (i 0))
    (lambda (in:double time:double channel:double data:double*)
      ;; every 11025 samples (i.e. 4Hz), do
      ;; - set pitch via random delay line length (100-1000 samples)
      ;; - fill delay line with white noise
      (if (= (% time 11025.0) 0.0)
          (begin (set! volume (random))
                 (set! pitch (+ 100.0 (* 900.0 (random))))
                 (dotimes (i 1024) (aset! dline i (random)))))
      ;; filter delay line in-place (only on first channel)
      (if (= channel 0.0)
          (aset! dline (dtoi64 (% time pitch))
                 (* 0.5 (+ (aref dline
                                 (dtoi64 (% time pitch)))
                           (aref dline
                                 (dtoi64 (% (- time 1.0) pitch)))))))
      ;; output (same for all channels i.e. mono)
      (* volume (aref dline (dtoi64 (% time pitch)))))))
```

Figure 7. A small, self contained, caprice written in Extempore

```
(bind-func poly-caprice 100000
  (let ((k1 (caprice 5000.0))
        (k2 (caprice 10000.0))
        (k3 (caprice 15000.0)))
    (lambda (in:double time:double channel:double data:double*)
      (* 0.5
         (+ (k1 in time channel data)
            (k2 in time channel data)
            (k3 in time channel data))))))
```

Figure 8. Caprice abstracted to multiple polymorphic parts.

values are used (along with a trivial implementation of Karplus-Strong) to generate the audio signal.

Our purpose here is to highlight the generality of specification afforded by working directly with the symbol system's floating-point *floor*. There are no explicit notes, unit-generators, schedulers, or any other sound or music related abstractions—the function simply returns the raw digital values which make up the audio waveform. What makes the example interesting is the high degree of musical information conveyed with little to no higher order musical abstractions (although our intention is still hinted at in our choice of symbol names such as `pitch` and `time`).

It is also worth noting the imperative nature of this code. This addresses two very real issues for livecoders. We suggest that imperative code allows audiences, as well as livecoders, to gain a greater insight into the operation of the algorithms being developed. Secondly, by working at "ground level" the livecoder is presented with considerably greater flexibility when exploring *new* algorithms.

Of course, we are not arguing against abstraction. Consider the simple change outlined in figure 8. By abstracting `caprice` into a higher order function we can trivially combine any number of polyphonic `caprice` *parts*. We also took the opportunity to introduce inter-onset times for each part.

It is in this easy switching between levels of abstraction, hoisting the tokens of the formal language up and down the ladder of musical meaning as required, that we see the true power of livecoding in a formal systems context. As an example, Extempore is fully committed to the idea of token generality by making the whole application stack available for run-time modification. For some perspective on the scope of this run-time modifability, Extempore's compiler, including the very semantics of the language, are available for run-time modification.

What this level of run-time reconfiguration means in practice is that composers are free to peek and poke their way around the whole audio stack, at run-time—replacing, extending or deleting the audio infrastructure as they see fit.

Breaking Open the Black Box

From a composer's perspective the desire to create abstractions is understandable, since music exhibits structure at so many different compositional layers.

> Since musical structures are architectonic, a particular sound stimulus which was considered to be a sound term or musical gesture on one architectonic level will, when considered as part of a larger more extended sound term, no longer function or be understood as a sound term in its own right. In other words, the sound stimulus which was formerly a sound term can also be viewed as a part of a larger structure in which it does not form independent probability relations with other sound terms. In short, the same sound stimulus may be a sound term on one architectonic level and not on another. (Meyer 1956, p.47)

Constructing high-level musical systems, in the form of algorithms which work on representations at music-theoretic levels (e.g. scale modes, beat-based meter, diatonic

```
(define alberti-bass
  (lambda (beat dur root)
    (play bass
    (alberti-arpeggiate beat root)
    80
    1/2)
    (callback (*metro* (+ beat (* .5 dur)))
        'alberti-bass (+ beat dur) dur
        (circle-of-fifths-next-root root))))

;; start the bassline, beginning on the tonic
(alberti-bass (*metro* 'get-beat 4) 1/2 'I)
```

Figure 9. An Extempore function which plays an alberti bassline as it moves through a circle of fifths.

harmony, etc.), does seem like an appealing use of an AFS from a compositional standpoint. However, as Gareth Loy points out:

> given a method or a rule, what is usually deemed compositionally interesting is to follow it as far as to establish a sense of inertia, or expectancy, and then to veer off in some way that is unexpected, but still somehow related to what has gone before. (Loy 1989, p.298)

High-level formal systems for composition may easily fall victim to the "iceberg effect", with tops visible above the water and large, unwieldy internals lying unseen below the surface. The common problem is that high-level musical algorithms tend to be either overly coherent or overly inventive—whereas it is not the aggregate of perceived coherence but a *distribution of coherence and invention through time* that is important for musical meaning (Meyer 1956). Coherence, within the context of music, is not simply a mathematical property, but a function of shared cultural and social values. Consider the code snippet in figure 9 (again in Extempore) of an alberti bassline as it moves harmonically through a circle of fifths, starting with the tonic.

```
(define alberti-bass-2
  (let ((scale '(0 2 4 5 7 9 11)))
    (lambda (beat dur root)
      (play bass
            ;; calculate which pitch to play by indexing into the
            ;; 'scale' list
            (+ 48 (list-ref
                   scale
                   (modulo
                    (+ root
                       ;; 'alberti' case statement
                       (case (modulo beat 2)
                         ((0) 0)
                         ((1/2 3/2) 4)
                         ((1) 2)))
                    7)))
            80
            1/2)
      (callback (*metro* (+ beat (* .5 dur)))
    'alberti-bass-2 (+ beat dur) dur
                ;; every four beats, move through the circle of 5ths
                (if (= (modulo beat 4) 0)
                    (modulo (+ root 3) 7)
                    root)))))

(alberti-bass-2 (*metro* 'get-beat 4) 1/2 0)
```

Figure 10. Another version of the alberti bassline function, this time using lower-level mathematical operations rather than high-level 'music composition' abstractions.

The `alberti-arpeggiate` and `circle-of-fifths-next-root` functions (which are part of a fictional high-level composition library) provide the arpeggiation and root movement information respectively, freeing the composer from the need to explicitly define these processes in their code.

Now consider another code snippet which produces the exact same musical result, but which prefers basic mathematical functions and programming language built-ins to higher-level musical abstractions. As discussed in the previous section, the flexibility in the musical meaning of the tokens in the source code allows easy switching between these different levels of abstraction.

In figure 10, `root` represents the scale degree (using a 0-based indexing scheme, so `0` for the tonic, `4` for the dominant, etc.). The `alberti-bass-2` function is called every half a beat (every quaver), and the exact pitch to play is determined by the current `root` plus an offset (calculated using a `case` statement) to perform the arpeggiation. The thing to note about this example is that although there is some musical domain knowledge encoded into (for instance) the `scale` list, the manipulation of the tokens is largely performed through basic mathematics. There is no domain knowledge hidden behind the tokens; no `perform-complex-musical-transformation` function hiding its internals.

In some senses, the first version (figure 9) is more transparent. The musically-savvy observer stands a good chance at guessing what the `alberti-arpeggiate` and `circle-of-fifths-next-root` functions do, and can therefore figure out what the overall sound is going to be. Again, this is similar to our ixi lang example from earlier, in which the pattern language version was more meaningful when considered purely as a string of tokens.

However, the livecoder is not simply appreciating the code as a string of tokens, they are listening, evaluating, and considering their next move. This is where the generality of the `alberti-bass-2` in figure 10 provides a benefit: the livecoder can tweak the case statement to change the arpeggiation pattern, or edit the `scale` variable to use a different mode, or alter the harmonic movement from a straight circle of fifths to something more complex. In the first `alberti-bass` example, in contrast, the very tokens which allow the composer to easily guess what the code *does* conspire to make it difficult to get it to do anything else. Without the ability to change the `alberti-arpeggiate` function (as would be the case if it were part of a monolithic formal composition system), the livecoder is limited in the changes that they can make.

In our own livecoding practice we have found this second approach to be more fruitful. It is a strategy which livecoding is relatively unique in affording: the ability to peer inside and manipulate the formal system while it is running, and to hear and judge the results of these manipulations instantaneously. While offline algorithmic composers must wait to hear how changes to their system are behaving, the livecoder is able to hear whether their system is working well (or not) much earlier, and is therefore able to apply corrective actions in a way which we have found to be extremely fertile from a creative standpoint.

Given the obvious temporal constraints imposed on the livecoder it may seem counter-intuitive to promote this lower-level structural approach. However, it promotes a generality which allows the livecoder to operate with a smaller subset of operators without sacrificing utility.

It is through a series of structural choices (the choice of symbolic floor, a flatter or more hierarchical structure) that an ontological commitment is made for a given performance. That these choices are essential to defining the character of a particular performance seems uncontroversial in the case of an improvisational practice like livecoding. However, we are also suggesting that this ontological commitment forms the basis of all musical composition. One ramification of this is that each individual computational work is inherently dependent on its own *unique* ontological commitments.

On Intention and Understanding

We have spent some time in this essay describing a triumvirate of musical meaning including the symbolic (code), the referent (sound), and the interpreter (both listener and machine). That musical meaning can be expressed as a variable combination of these constituent parts is of some interest to the livecoding community whose mantra is that

"Code should be seen as well as heard" (The "Lubeck 04 Manifesto" Ward et al. 2004).

Audiences believe in the logic and purposefulness of the composer and his intentions. As Leonard Meyer points out "Though seeming accident is a delight, we believe that real accident is foreign to good art" (Meyer 1956, p.74).

The variability of relationships between a particular livecoding performance's meanings are a reflection of real objective cultural values. Cultural values that express themselves in the form of a musical style, community or movement. To quote from Leonard Meyer:

> Musical meaning and significance, like other kinds of significant gestures and symbols, arise out of and presuppose the social processes of experience which constitute the musical universes of discourse. (Meyer 1956, p.60)

However, it is clear that musical intention and musical understanding do not form a fixed and constant relationship. Where art is at its most powerful is in the margins—the space between total understanding and complete intention. Nevertheless, there must always be enough shared understanding for communication to remain possible. It is in finding the correct balance between norms and deviants that artists struggle. For Leonard Meyer, musical meaning is a product of these expectations.

The ability to balance the norms and deviants required to communicate a meaningful musical message has proved to be problematic for purely formal computational systems. We believe that by giving the responsibility of higher-level structural coherence (through the orchestration of runtime processes) to the livecoder, human perception and intuition can be brought to bear on what is ultimately a cultural and inherently non-linear problem. The livecoder is able then to choose a meaningful pathway between social norms and deviants, and most importantly to chart this path anew for each and every performance.

> While the meaning of a musical work as a whole, as a single sound term, is not simply the sum of the meanings of its parts, neither is the entire meaning of the work solely that of its highest architectonic level. The lower levels are both means to an end and ends in themselves. The entire meaning of a work, as distinguished from the meaning of the work as a single sound term, includes both the meanings of the several parts and the meaning of the work as a single sound term or gesture. Both must be considered in any analysis of meaning. (Meyer 1956, p.47)

Ultimately though it is arguably the support that livecoding provides for easily shifting *between* AFSs of different levels - different semantic layers - that may prove its enduring legacy. As Meyer articulates in the previous quote, musical form is a complex interrelationship of hierarchical meanings that are not easy to untangle. The great advantage for livecoding is the presence of a human agent who provides an exit/re-entry point for switches between formal systems as well as for the redefinition of a formal system's rules and axioms on-the-fly. This human-in-the-loop approach to the development of formal systems is a unique contribution to the artistic landscape.

Conclusion

The livecoder's ability to orchestrate abstract formal processes in perceptual response to the acoustic environment provides scope for intuition and play. By supporting a dynamic interplay between the composer's formal intentions and the machine's formally derived actions, the composer is able to guide the musical outcome, as embodied in the physical environment. By placing a human in-the-loop, livecoding provides not only the means to critique an algorithm (as any offline method also allows) but also to modify an algorithm over time—to *steer* the result in culturally meaningful directions.

In this essay we have attempted to open a dialogue on the multiple levels of meaning present in livecoding practice. We have discussed the composer's role in the formation of various ontological commitments, with some regard to the inevitable compromises associated with different levels of representation. Ultimately we have only begun to explore the complex interwoven semantics inherent in livecoding practice. Our hope then for this modest contribution is to engage the community in a robust discussion surrounding the many meanings of livecoding.

We conclude with an observation from William Schottstaedt in 1987. In regards to his PLA computer music language, he wrote

> To my surprise, neither the real-time input of data nor the real-time interaction with composing algorithms has generated much interest among other composers.
>
> (William Schottstaedt, quoted in Mathews and Pierce 1989, p.224)

We believe that after ten years of livecoding practice the value of interacting with composing algorithms in real-time is beginning to reveal itself, and in ways that the computer music community of three decades ago could not have imagined.

References

Biles, J. A. 2007. "Improvizing with Genetic Algorithms: GenJam." In *Evolutionary Computer Music*. London: Springer London, pp. 137–169.

Boretz, B. 1970. "Nelson Goodman's Languages of Art from a musical point of view." *The Journal of Philosophy* 67(16):540–552.

Collins, N., A. McLean, J. Rohrhuber, and A. Ward. 2003. "Live coding in laptop performance." *Organised Sound* 8(03):321–330.

Cross, I., and E. Tolbert. 2009. "Music and meaning." In *The Oxford handbook of music psychology*. Oxford University Press Oxford, pp. 24–34.

Dennett, D. C. 1989. *The international stance*. Cambridge, Massachusettes: MIT Press.

Eco, U. 1979. *Theory of semiotics*, volume 217. Bloomington: Indiana University Press.

Edwards, M. 2011. "Algorithmic composition: computational thinking in music." *Communications of the ACM* 54(7):58–67.

Emmerson, S. 1986. *The language of electroacoustic music*. London: Macmillan.

Essl, K. 2007. "Algorithmic Composition." In C. . J. d'Escrivan, (editor) *The Cambridge Companion to Electronic Music*. Cambridge University Press.

Fux, J., and A. Mann. 1965. *The study of counterpoint from Johann Joseph Fux's Gradus ad Parnassum*, volume 277. New York: W. W. Norton & Company.

Goodman, N. 1976. *Languages of art: An approach to a theory of symbols*. Cambridge, Massachusettes: Hackett Publishing Company.

Haugeland, J. 1981. "Semantic engines: An introduction to mind design." In J. Haugeland, (editor) *Mind Design*. Cambridge, Massachusettes: MIT Press.

Loy, G. 1989. "Composing With Computers — a Survey." In M. V. Mathews, and J. R. Pierce, (editors) *Current directions in computer music research*. Cambridge, Massachusettes: MIT Press.

Magnusson, T. 2011a. "Algorithms as Scores: Coding Live Music." *Leonardo Music Journal* 21(21):19–23.

Magnusson, T. 2011b. "ixi lang: a SuperCollider parasite for live coding." In *Proceedings of the International Computer Music Conference*. University of Huddersfield.

Mathews, M., and J. Pierce. 1989. *Current Directions in Computer Music Research*. Cambridge, Massachusettes: MIT Press.

McLean, A., and G. Wiggins. 2010. "Bricolage programming in the creative arts." *22nd annual psychology of programming interest group* .

Meyer, L. 1956. *Emotion and meaning in music*. Chicago: University of Chicago Press.

Morris, C. W. 1938. *Foundations of the Theory of Signs*, volume 1. University of Chicago Press.

Rohrhuber, J., and A. de Campo. 2009. "Improvising Formalisation-Conversational Programming and Live Coding." .

Rohrhuber, J., A. de Campo, R. Wieser, J.-K. van Kampen, E. Ho, and H. Hölzl. 2007. "Purloined letters and distributed persons." In *Music in the Global Village Conference (Budapest)*.

Smith, B. C. 1996. *On the origin of objects*. Cambridge, Massachusettes: MIT Press.

Sorensen, A., and H. Gardner. 2010. "Programming with time: cyber-physical programming with impromptu." *ACM SIGPLAN Notices* 45(10):822–834.

Wang, G., and P. R. Cook. 2004. "On-the-fly programming: using code as an expressive musical instrument." In *Proceedings of the 2004 conference on New interfaces for musical expression*. National University of Singapore, pp. 138–143.

Ward, A., J. Rohrhuber, F. Olofsson, A. McLean, D. Griffiths, N. Collins, and A. Alexander. 2004. "Live Algorithm Programming and a Temporary Organisation for its Promotion." In O. Goriunova, and A. Shulgin, (editors) *read_me — Software Art and Cultures*.

Zemanek, H. 1966. "Semiotics and programming languages." *Communications of the ACM* 9(3):139–143.