STRANGE LOOPS IN CFML, A LIVECODER'S RIDDLE

Adam M. Smith
Santa Cruz, CA, USA
adam@adamsmith.as

ABSTRACT

The practice of livecoding borrows heavily from the techniques and vocabulary of music and computer programming. In a setting where the design, implementation, execution, and reflective redesign of software systems are simultaneously overlaid and entangled with sonic creativity in the moment, traditional vocabularies fail to offer more than ambiguous metaphor. This paper uses examples from *cfml*, a minimal livecoding system, to probe the livecoder's conceptual landscape, revealing unfamiliar structures and processes reminiscent of Hofstadter's strange loops. Results from even rudimentary practice with an intentionally impoverished tool point at the need for further inquiry into these natively-livecoding concepts that are at the very edge of expression within the terminology of livecoding's computer music origins.

1. INTRODUCTION

Canonically, livecoding challenges an artist/programmer to interactively develop a software system that will generate musical entertainment for an audience in an real-time fashion. At a relatively obscure extreme of computer music, practitioners of livecoding are left to adopt and awkwardly apply loan words from the parent disciplines of music (e.g. "composition") and engineering (e.g. "program"). In this work, my goal is to highlight the unique structures and processes inherent and native to livecoding that go undescribed with traditional vocabularies.

In a recent *Leonardo Music Journal* article [7], Thor Magnusson draws on experience with his own livecoding language (ixi lang) to reflect on the traditional conception of "score" as a "message from a composer to an instrumentalist" in light of livecoding practice. Finding historical precedent for scores as a musical technology and as a mnemonic device, Magnusson offers algorithms as the natural progression of traditional graphic scores. Admitting algorithms as scores generalizes the idea of a score as a set of steps to follow to determine how the literal notes of a performance should be realized.

When algorithms are defined and redefined in an *on-the-fly* manner [11], it quickly becomes difficult to point out exactly where is the score for a particular piece. Some algorithms may be at work producing a stream of notes for real-time synthesis while others may adjust how high-level patterns are interpreted and expanded. When all of these algorithms are available for interactive and expressive redefinition, how do we determine the object of "composition" or "performance"? Is it a piece of music, the score for a piece of music, the procedure for generating a score, the interpreter for a language designed for describing score-generating procedures? As most livecoding systems are tools for making tools in addition to simply making music, Magnusson notes that they have a "self-reflexivity" property that suggests that any or all of the above may legitimately answer what is being composed, performed, or (more simply) livecoded.

In all but the simplest livecoding performances, the sequence of notes to be synthesized is not uniquely prescribed by deterministic rules, a product of both aleatoricism and indeterminacy. Bringing in computers as the interpreters and performers of music from rigidly-defined rule systems takes us through algorithmic composition to generative music [5], however livecoding has something that goes beyond far this, something difficult to express in our available musical and engineering terminology.

Livecoding performances are (to single out just one feature) pathologically indeterminant. The flavor of a livecoding piece hinges not only on *what* new code is injected during performance but *when* and *where* that injection occurs with respect to the the sonic and computational state of the piece. If the audio stream computed in generative music is only an epiphenomenal shadow of an underlying algorithm, then the succession of ephemeral algorithms at play in a livecoded music piece are but shadows of something we have yet to name.

Hofstadter sketches a *strange loop* as "a paradoxical level-crossing feedback loop" or when "there is a shift from one level of abstraction (or structure) to another, which feels like an upwards movement in an heirarchy, and yet somehow the sucessive 'upward' shifts turn out to gives to a closed cycle" [6, p. 102].

At the heart of livecoding is one such strange loop: the livecoder's instantaneous choice of algorithm depends on how the music is being performed and received by an audience, and how the music is being performed and received depends on the choice of algorithm. Solutions to this cyclic determination are unstable: springing into existence from silence, oscillating unpredictably, and eventually decaying without any causes that is not equally perceivable as an effect. How the music emerges from the algorithm is easily recognizable as the domain of generative music, but how reactions to this music turn into code-splices or how those splices achieve artistic effect is all but unknown.

In this paper, I use examples from my livecoding system cfml [9] as a way to point at those strange structures and processes that seem to be at the core of livecoding. In section 2, I review cfml as a generative music system and explain how a series of small, unconscious choices during its implementation led to it gaining the livecoding nature. In section 3, I give a walkthrough of the virtual machine at the heart of cfml's performance engine to exemplify the unfamiliar processes that the system requires a programmer to reason through. Finally, in section 4, I describe a sequence of three études that portray cfml as a trivial generative music system, an expressive live performance tool, and a curiously complex artifact beyond the understanding of its creator.

2. PROGRAMMING IN CFML

At first glance, cfml is a straight-forward generative music programming language. Where cfml diverges from the languages that inspired it (described later) is in its use of a *strange* construct without clear precedent in other languages. The key structure that every expression in cfml assembles is that which the programmer composes, that which composes the music heard, and that which is composed of other instances of the same type: a data/code structure that I call (for lack of a more transparent name) a *comp*. (This is not intended as a universal concept for computer music; comps are simply the peculiar creatures that occur in cfml.) To understand the accidental emergence of comps and their unexpected affordances for live performance, it is important to understand the original intent of cfml and its surface features.

2.1. Development Context

Originally developed for use as a supporting example in a lecture for an undergraduate computer science audience, cfml was conceived in late 2009 as the Context Free Music Language. Cfml was to be a lightweight sonic analog of Chris Coyne's cfdg [4] (a language and interpreter for Context Free Design Grammars in the domain of two-dimensional visual art) and Mikael Christensen's Structure Synth [3] (a three-dimensional adaptation of cfdg).

The grammar constructs used in each of these tools are derived from shape grammars, a concept from architecture [1]. Where shape grammars were historically used descriptively (as a way to reflect on spatial decomposition and the reuse of geometric motifs), the intended use of grammars in cfdg and Structure Synth is the generation of new artifacts. As such, these systems include pragmatic features (such as context-sensitive, automatic termination of would-be infinite recursion) that distance them from their context-free namesakes in the Chomsky hierarchy of formal languages [2].

The analogy between the visual and sonic arts embodied by cfdg and cfml is by no means original or all-encompasing (e.g. there is no obvious sonic equivalent of geometric rotation, nor is there an unambiguous equivalent for temporal succession), however it was deep enough

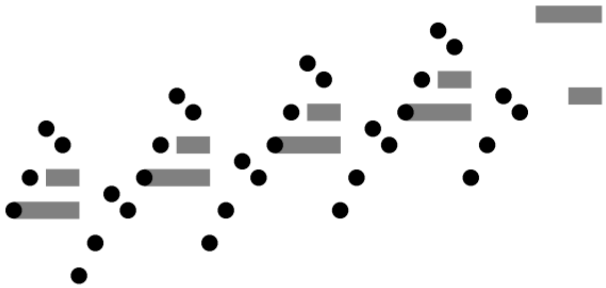

Figure 1. A sample output from a cfdg program, the visual analog of the cfml étude in subsection 4.1 entitled *The Escalator*.

to serve the purpose of illustrating the abstract concepts of recursion and nondeterminism in a sensorial yet medium-agnostic manner. Where cfdg arranges geometric primitives (circles, squares, and triangles) on a two-dimensional drawing canvas, cfdg arranges musical primitives (equivalent to MIDI note events) on a one-dimensional performance timeline. Though this article does not hang on understanding the analogy between the two languages, the reader is invited to relate an example piece in cfdg (output sample shown in Figure 1) with the instructions for performing the cfml étude described in subsection 4.1.

To speed development of my new programming language for generative music, I opted to implement cfml as a embedded domain-specific language within Andrew Sorensen's Impromptu [10], a language and environment for audio-visual livecoding. This choice provided me with a convenient performance interface, a familiar and well-defined surface syntax (that of Scheme), and an interactive interpreter that was already integrated with a real-time synthesis framework. In a bit of lazy programming, I decided that elements of a musical composition would not be evaluated until just before they were needed. This single implementation detail (an unconscious reflex from livecoding practice in Impromptu with no special connection to grammars) would imbue cfml with a very special property: the ability to reactively change the rules of composition while a piece was concurrently being composed and performed.

2.2. Literals

The generation of music in cfml starts with literal notes. The simplest type of comp holds a small collection of notes to be played back through Impromptu's scheduling and synthesis back end. Such a comp is constructed using the `literal` keyword:

```
(literal duration note-descriptors)
```

The duration of a note bundle is expressed in beats and the notes themselves are composed of 5-element lists `(o c p v d)` where `o` is a relative onset time (in beats), `c` is a MIDI channel, `p` is a relative pitch number (used to compute an absolute pitch number given a separately

specified scale), `v` is a relative velocity, and `d` is relative duration (also in beats).

Comps are performed using the `perform` command, passing the comp, a tempo (in beats-per-minute), and a scale (a list of relative MIDI pitch offsets within each octave). This expression plays a single half-second note on middle-C through whichever synthesizers are responding on MIDI channel 1.

```
(perform (literal 1 '((0 1 0 1 1)))
         120
         (pc:scale 0 'dorian)))
```

2.3. Procedures

To simplify larger expressions, Scheme's `define` operator can be used to bind comp values in the interpreter's global symbol table. This first expression gives the *result* of our expression a concise reusable name:

```
(define beep
  (literal 1 '((0 1 0 1 1))))
```

This slight variation has radically different semantics (note the parentheses):

```
(define (beep)
  (literal 1 '((0 1 0 1 1))))
```

In the first version, `beep` is bound to the comp returned by `literal`. In the second version, `beep` is bound to a *procedure* that will return a comp when invoked. For consistency purposes, comp-returning procedures are also considered comps. In most cfml programs, nearly all definitions are of this second form: defining Scheme procedures that take no arguments (making them context free, in a sense). Being able to delay the evaluation of the body of a procedure until a later time (during which various other global names have been redefined) is critical for livecoding uses of cfml.

2.4. Modifiers

Hand-entering the parameters for a note bundle is tedious. Thus, cfml includes operators for modifying how a predefined comp will be performed. Currently, transposition, velocity (volume) scaling, and duration (tempo) scaling are the only modifiers:

```
(tra shift comp) ; transpose by shift
(vol ratio comp) ; scale velocity by ratio
(dur ratio comp) ; scale duration by ratio
```

This expression describes a comp that will perform `beep` at triple length, half volume, and shifted up by two pitches in the working scale:

```
(tra 2 (vol 1/2 (dur 3 beep)))
```

2.5. Combinators

Cfml has three operators (called combinators) for building more complex comps from simpler ones: `after`, `during`, and `choose`. Like modifiers, expressions using combinators can be nested arbitrarily. The following expression yields a comp encoding the nondeterministic performance of three possibilities (a triple-beep, a short chord, or a long beep):

```
(choose (after beep beep beep)
        (dur 1/2 (during beep
                         (tra 2 beep)
                         (tra 4 beep))
        (dur 2 beep))
```

Like the other combinators, `after`, is an operator that can be applied to any number of comps. As the name suggests, the comp returned by an `after` expression will result in the performance of the first element of its arguments followed by successive performance of the remaining elements. For the purposes of sequencing, the duration (first argument) to a `literal` comp is used to determine the length of a note bundle (allowing for bundles with silence or notes that extend beyond the nominal boundaries of a local pattern).

The `during` combinator yields similar results to `after` except that all comps are performed in parallel instead of in a series. The performance length of a comp produced with `during` is the length of the longest comp it performs.

Finally, to allow the expression of aleatoric composition rules, the `choose` operator accepts any number of comps as arguments and will decide randomly (with a uniform distribution) which one to perform. The length of a comp resulting from `choose` is the length of the randomly chosen comp it performed.

2.6. Live Execution State

At this point, I have introduced all of cfml's surface features. Any comp can be produced by some combination of literals, modifiers, combinators, and procedure definitions. Fluent performance with this language, however, requires understanding more. During live execution of a cfml program inside of Impromptu, there are four primary types of program state that can effect either how audio is synthesized or which program edits a programmer should consider making to achieve a certain effect.

The first type of execution state is the queue of notes to be played. This data structure, holding a set of fully-resolved MIDI events, is maintained by Impromptu and is not directly visible to the programmer. Once notes enter the queue, they are committed for synthesis and cannot be directly affected by the programmer from within cfml.

The next type of execution state is also an invisible, internal queue. When a comp is delayed in time (through the use of an `after` combinator), a reference to that comp and its intended execution environment are saved in a data structure associated with Impromptu's callback scheduler. While the programmer cannot easily un-schedule future performance tasks, the programmer does have direct control over the environment in which these future tasks will run. Because of the use of procedure definitions to delay computation, one version of a comp can be made to refer

to a future version of itself (a state unheard of in conventional programming tasks).

The third and most obvious type of live execution state is the global symbol table (the structure that stores the mappings created by `define`). By re-evaluating a modified version of a definition, the newly defined comp will be made available to any procedures attempting to look up the old comp by name (though it is also possible to retain a reference to the older version of the definition, a fact exploited in the étude described in subsection 4.3)

The final (and often overlooked) type of execution state is the state of the programmer's text editor. Even though this state exists outside of composition rules in play at any time, it is still data stored in Impromptu's memory. Knowing which of the definitions visible on screen is still valid with respect to the global symbol table takes programmer effort (though an alternate graphical interface might ease this burden). When considering a redefinition of an existing procedure definition, the programmer must choose between destructively editing the current definition in place (losing access to the old version) or writing a new definition below (possibly with the help of copy/paste) so that the old definition can be restored or re-edited with ease. Opting to leave more code-at-hand requires time to create new code, consumes valuable screen-space while performing, and increases the cognitive burden of remembering which definitions are active when there are more to choose between.

Mentally tracking each of these types of execution state and using that knowledge to make strategic decisions about what with, how, and when to redefine names in the global symbol table seems to be a skill unique to livecoding.

3. UNPACKING PERFORMANCE

In the preceding discussion, I have referred to composition, programming, performance, and execution as if they were nearly separable concepts. In livecoding, however, this separation can never be complete. In this section, I aim to give an accurate description of the distinct artistic and technical processes that are carelessly attached to overloaded terms when talking about cfml at a high level.

When I use cfml as notation, an instrument, or a prop for performance, I am referring to performance as the process of being a livecoder in the moment of play. Second-by-second, my performance will consist of creating and naming comps. When I use cfml as a compiler, interpreter, synthesizer, or computation engine, I'm referring to the performance as in the generative music mode, where a virtual machine is consuming comps as machine instructions, performing the operations they describe, and yielding result data (some of which is directed to a real-time synthesizer).

As comps are the glue between these two very distinct senses of performance in cfml, I should describe how they are created and consumed in more detail.

3.1. Comps as Data Structures, Code Structures

Having initially gotten cfml to a functioning state in an exploratory mode of software development, I must admit that comps, as the central representation structure in my system, are more the product of accident than conscious design. In a Scala-based reimplementation of the core ideas of cfml, I took the opportunity to reflect on what a comp is and what it means. After this, I best understood comps as data structures that describe a single action to be performed by a strange kind of virtual machine (VM). This reading is supported by cfml's interpreter being organized as an opcode dispatch loop (albeit for a slightly different language than the programmer sees on the surface).

When a cfml expression such as `(after beep (tra 2 beep)` is evaluated (as if it were no different from any other Scheme expression) the result is actually a graph of comps. It is this graph (a root comp and all of the other comps it points to, transitively) that is actually interpreted by the cfml VM. Depending on the type of the comp, the action performed is sometimes better read as code compilation, high-level composition work, or low-level score realization (i.e. even at the machine level, performance is not a simple concept).

Comps can be read as instructions for how and when to compose new musical material; they are data structures that can be interpreted to yield synthesizable notes. Contrast this with the cfml expressions seen throughout the paper so far. None of these were really programs for composing music; they were programs for composing comps (in the sense of nesting parts to form wholes). Comps are not just scores or rules for generating scores (one metalevel up), but potentially rules-for-rules-for-rules-. . . -for generating scores at an ambiguous metalevel. Cfml expressions say one thing and do another, but seem to end up doing what they say eventually. Understanding the mixing meta-levels of languages within languages and interpreters for interpreters is another facet of the riddle cfml unintentionally presents to those who would use it (and similar systems) fluently.

3.2. VM Instructions

There are six types of comps: four have a relatively clean mapping to cfml's surface language, one is an understandable abstraction, and the final type stretches the limits of the VM-metaphor for cfml's operation.

The internal state of the cfml VM is encoded in seven registers: `score` (a reference to a single comp to be performed), `tempo` (set in `perform` and altered by `dur`), `scale` (set in `perform`), `root` (offset in scale altered by `tra`), `time` (logical time in samples that this comp should be performed, incremented by nominal durations), `volume` (velocity scaling ratio altered by `vol`), and `k` (a continuation to be called when performance of the current comp completes).

3.2.1. *op-literal*

Unsurprisingly, `op-literal` instructions are produced by expressions using the `literal` operator. That is, when I evaluate `(literal 1 '((0 1 0 1 1)))`, the result I see is a new list data structure: `(op-literal 1 ((0 1 0 1 1)))` (no sound is produced until this VM instruction is passed to `perform`, the function that kicks off temporally recursive VM execution).

To execute an `op-literal` instruction, the VM interprets the note descriptors in the context of the current values of the VM registers (mapping relative pitch to in-scale absolute pitch, determining duration in samples from the current tempo, etc.) and passes the resulting data to Impromptu's MIDI event scheduler. Knowing the exact duration of the literal data, the VM uses Impromptu's `callback` command to schedule the continuation `k` (which will begin performance of any comps that might have been scheduled after this one) for the appropriate delay.

3.2.2. *op-after*

Evaluating an expression like `(after beep beep beep)` results in a value like `(op-after #<beep> (op-after #<beep> #<beep>))` where `#<beep>` is a reference to the current value of `beep` in the global symbol table. That is to say, the `after` combinator dynamically translates its list of arguments into a chain of two-argument `op-after` instructions.

To execute an `op-after` instruction, the VM recursively executes the first argument to the instruction, passing the current continuation (extracted with `call/cc`) for the new value of the `k` register. This way, when the child instruction completes performance, the VM will resume execution in the state it had upon entrance into this instruction (any transpositions or volume adjustments made in a subordinate comp cannot affect the parent). Upon completion of performance of the left child, the right child is executed.

So far, I have just followed the standard idiom for delayed computation in Impromptu: temporal recursion [10]. The essence of temporal recursion is to perform some operation right now and then schedule a delayed callback to your own code, passing only the remainder of the work to be performed later. To overcome jitter and computation delays, it is conventional to schedule a temporally recursive call a brief moment ahead of when the first notes created by that computation might start playing (cfml uses an offset of 100ms).

3.2.3. *op-during*

The relation between `during` expressions and the `op-during` VM instruction is analogous to the relation between `after` expressions and `op-after` instructions. The only twist is that, instead of waiting for the left child of an `op-during` to complete before executing the right child, both are immediately scheduled for execution in parallel. Once both children have completed (in any order), the execution/performance of the `op-during` instruction is considered complete.

3.2.4. *op-choose*

Comps created with the `choose` combinator (`op-choose` instructions) are simple enough to execute: a random argument of the instruction is selected and then that instruction is recursively executed.

3.2.5. *op-tweak*

To capture the environment adjustment (register tweaking) required by the `tra`, `vol`, and `dur` modifiers, a single type of instruction suffices. Evaluating an expression like `(tra +2 beep)` results in a comp like `(op-tweak #<fun> #<beep>)` where `#<fun>` is a dynamically created function that takes the current assignment of the note-parameter related VM registers and computes a new value for each of them.

To execute an `op-tweak` instruction, the tweaking function is applied to the current VM registers and the child comp is recursively executed by a VM in the newly described state.

3.2.6. {*procedure*}

The final type of instruction is not associated with an opcode tag. Recall that we previously defined procedures as a kind of comp. Execution of comps that are really procedure references is trivial: run the procedure (assumed to take no arguments) and recursively execute the comp it returns.

When used as part of a livecoding piece, the only kind of procedures passed to the cfml VM are procedures defined by cfml surface language expressions. So, when the VM executes this type of comp, it is actually twisting back a meta-level to ask the outer Scheme interpreter to dynamically compile programmer-entered expressions down to the VM's language (another strange loop), potentially collecting up recently redefined values out of the global symbol table.

3.3. Human Input

Armed with a better understanding of what it means for the cfml VM to perform a comp (a stretched sense of machine instruction execution), I should return to the sense of performance in being a human livecoder, in the moment.

I have shown that the programmer's input is clearly scoped to putting new procedure definitions into named slots in the global symbol table. This operation is non-destructive and cannot modify any comp that is directly referenced by another comp. Outside of inspecting the global symbol table, the VM is insulated from the programmer's activity.

Consider the evaluation of this snippet of cfml (the focus of subsection 4.2):

```
(define (song) (after phrase song))
```

The symbol `song` now points to a procedure. This procedure, when executed, will produce an `op-after` instruction with the procedures currently pointed to by `phrase` and `song` in the global symbol table as its two arguments (effectively taking a snapshot of the supporting composition rules active at that point in time). With no other changes, execution of `(song)` always produces identical results. However, if `song` is redefined during the performance of `phrase`, the new value of `song` will be picked up when the VM executes the instruction's second argument in an attempt to resolve it into an concrete instruction. This is not just a recap of instruction execution; it is an example of the mental code-tracing process that is required to decide when to evaluate a redefinition in order to get it picked up on the appropriate musical cycle.

This required thinking through future-self-reference, yet another strange loop, is simultaneously at the core of what makes livecoding unsettling from traditional programming perspectives and what makes it comprehensible as a medium for live musical expression. It is indeed true that `song` plays *after* `phrase` (just as the surface language code said it would), but it does it as part of a cyclic determination process that wraps up even the livecoder's internal thought processes!

The ultimate meaning of `op-after` instructions is tied to meta-circular interpretation of self-referential structures that are generated on the fly from interactively modified definitions. This situation is clarified little by the use of continuations in the VM's implementation (continuations are a programming language feature often explained with reference to time travel [8]).

Despite this wild complexity, the affordances for human expression are clear: redefining `phrase` will alter the future of the current piece with the effects becoming immediately audible after the completion of the current version of `phrase`.

4. TROIS ÉTUDES

Explorations of conceptual landscapes aside, the nominal purpose of cfml is to allow a livecoder to entertain an audience with some simple, synthesized music. In this section, I describe three elementary performances that provide a concrete realization of cfml's expected and not-so-expected features.

To simplify the examples below, imagine the following definition has been evaluated before any performance:

```
(define (run comp)
  (perform comp 120 (pc:scale 0 'dorian)))
```

4.1. *The Escalator*, a Disposable Ditty

My first example, *The Escalator*, is a rather unreliable mechanized stairway. See Figure 1 for a visual analog of the music generated by the definitions below. This piece exhibits only the basic generative music faculties of the language, and it can be performed without interactive livecoding. It was adapted from the self-contained piece, included with the cfml source distribution, that plays once after all of the core definitions are loaded.

First, I define a generic bump of the MIDI keyboard for use as a building block in larger comps:

```
(define (bump)
   (literal 1/2 '((0 3 0 1 1/2))))
```

Performing this comp (a comp-generating procedure), by evaluating `(run bump)`, yields a quarter-second guitar pluck on middle-C. Sequencing four transposed copies of this pattern into a more interesting unit requires the following definition:

```
(define (lump)
   (after bump
          (tra 2 bump)
          (tra 5 bump)
          (tra 4 bump)))
```

Before assembling the lumpy stairway, I define two patterns for the string instrument on another channel.

```
(define (string-step)
  (literal 2 '((0 4 0 1 2)(1 4 2 1 1))))

(define (string-end)
  (literal 2 '((0 4 4 1 2)(1 4 -1 1 1))))
```

Throwing caution to the wind, I define a single future-self-referential comp that describes choosing between the option of immediately ending the piece and the option of the up-transposed sequence of an interesting phrase with the future performance of the same comp. In Scheme, `let` is a construct that enables binding of local variables.

```
(define (song)
  (let ((phrase
         (during (vol 3/4 string-step)
                 (after lump
                        (tra -4 lump)))))
    (choose (vol 2/3 string-end)
            (tra +2 (after phrase
                           song)))))
```

Evaluating `(run song)` yields, 50% of the time, simply performance of the `string-end` pattern. However, repeated attempts will often reveal ascending chains of notes that rise in pitch (inadvertently sonifying the how many times in a row I can flip heads on a fair coin).

Had cfml been implemented as a batch generation system, performances of *The Escalator* would sound no different. In this piece, the strange loops in cfml lie dormant, masquerading as a harmless recursive decomposition of a generative music piece.

4.2. *The Noodle*, a Livecoded Jam Session

The obvious edge livecoding has over generative music is that the rules of composition are flexible; the livecoder can adapt these rules to the evolving tastes of a live audience. In *The Noodle*, I demonstrate how repeating a very simple phrase can provide the foundation for many moments of enjoyment when the conditions of that repetition are actively steered by the programmer.

I begin with a staccato note on the piano synthesizer:

```
(define (blip)
  (literal 1/4 '((0 6 0 1 1/4))))
```

As in *The Escalator*, I build the overall composition on a four-note ostinato:

```
(define (phrase)
  (after blip
         (tra 4 blip)
         blip
         (tra 9 blip)))
```

The initial song structure is the staid trope of temporal right-recursion:

```
(define (song) (after phrase song))
```

At this point, I evaluate `(run song)` to kick off live performance of the definitions thus far. This begins a stable arpeggio that can hold the audience's interest for perhaps a few iterations. Left unattended, it would continue to self-referentially unfold for eternity. To keep the piece alive, I begin an upward progression with this destructive, in-place redefinition:

```
(define (song)
  (after phrase (tra +1 song)))
```

The interest created by the introduction of an upward trend turns to tension as the notes creep into uncomfortably high register. Again, I step in to rescue the piece from boredom and increase drama with another redefinition, this time starting a much steeper downward trend:

```
(define (song)
  (after phrase (tra -2 song)))
```

By now, the trick of steering the noodling melody up and down by tweaking the transposition parameter is becoming clear. As the notes descend, I begin work on factoring out a tweakable knob and catch the recursion just before it enters an uncomfortably low register, reinstating the gentle upward trend with this refactored definition:

```
(define delta +1)
(define (song)
  (after phrase (tra delta song)))
```

From here, sharply tweaking `delta` between `-3` and `+3` can create a few more moments of interest, but the direct control requires too much attention to maintain while thinking through larger scale flourishes. Capturing and automating my manual steering with a random walk is simple enough:

```
(define delta +1)
(define (song)
  (after phrase
         (choose (tra (+ delta 2) song)
                 (tra (- delta 2) song))))
```

With the current algorithms creating some complexity on their own, I need rarely tweak `delta` except for when the piece wanders too far off course. Between these sparse tweaks, I am free to adjust the spread of notes in `phrase`, add an alternative `phrase2`, build larger phrases with interactively-swappable substructure, and begin to automate incremental variation of the dynamics (note velocity) using a similar random-walk scheme.

At no time during the performance of this piece is there ever a particularly interesting algorithm executing. Further, most of the algorithms encountered, when left unattended even for a few seconds, would quickly be producing notes with pitches outside the acceptable range for synthesis. The programmer's artistic gestures are encoded in exploratory vacillations between moments of intimate, interactive control and algorithm-moderated distance while the next big compositional twist is being conceived and encoded without audible feedback.

In *The Noodle*, the downward causality inherent in cfml's strange loops becomes apparent. A bottom-up analysis of definition such as `(define (song) (after phrase song))` would suggest that any interestingness in this piece must come from `phrase`–it is the only part of definition that apparently references concrete notes. However, as I have demonstrated, the interestingness of this piece comes from the future-self-reference to `song`. The particular moves I make in redefining the future of the piece are a function of how it has unfolded so far and how that unfolding resonates with the audience. The phrase-generating algorithms never have any appreciable depth; depth is inherited from contrast with as-yet-to-be-defined future versions of the `song` comp. That is to say, `(define (song) (after phrase song))` expresses a template for a piece of music that (however implicitly) includes the reflective programmer and reactive audience as part of the generative loop.

4.3. *The Bent Pyramid*, a Mild Bludgeoning

The Escalator introduced the surface of cfml in an non-interactive setting, and *The Noodle* demonstrated how to fluently wield redefinitions to maintain an audience's interest far beyond the limits of algorithms expressible in the language's intentionally-limited model of computation. *The Bent Pyramid*, however, is not so pleasant. It jumps to an extreme of what this language designer can comprehend, raising many questions and answering few.

As usual, the piece begins with a single literal comp:

```
(define (pluck)
  (literal 1/2 '((0 2 -14 1 1/2))))
```

The core definition below is a symmetric variation on the structure in *The Noodle*. Note that `pyramid` refers to itself, but not as last item of the `after` combinator. This definition is recursive, but is not right or tail-recursive.

```
(define (pyramid)
   (after pluck
          (tra 2 pyramid)
          (tra 1 pluck)))
```

Beginning live synthesis with `(run pyramid)`, the audience hears a sequence of plain notes ascending the scale. Not long after, the unbounded recursion nears the

uncomfortably high register avoided in previous pieces. Slowing (but not halting) progress into this foreboding territory, I redefine `pluck` to take four times as long:

```
(define (pluck)
  (literal 1/2 '((0 2 -14 1 1/2))))
```

When the pitch becomes unbearable, I interject:

```
(define pyramid pluck)
```

This non-recursive definition would have stopped a right-recursive piece dead. However, in *The Bent Pyramid*, it signals the apex. When this new rule is picked up, the sequence of notes continues, creeping downward.

As time progresses, the pitch eventually reaches the point where I adjusted the definition of `pluck` on the way up. At this point, with no manual intervention, the tempo unexpectedly quickens to the original pace, continuing to descend in pitch. Then, right when the sequence would have passed the pitch at which the bent pyramid had begun construction, the piece terminates (again, without intervention).

How is this possible? Recall that when a comp-generating function is evaluated, it takes a snapshot of the current working definitions of the comps it references. When `pyramid` is evaluated very early in the piece, the comp it returns stores a reference to the fast-tempo variant of `pluck`, and this is the version it uses when performing the third term seen in the `after` expression. Likewise, when `pyramid` is redefined to be non-recursive, this does not modify any comps that are currently in the middle of performance (all those created in ascending the pyramid that are currently packaged up inside of continuations).

What does this imply? Fluent livecoding, not just with cfml but any livecoding system that includes closures (procedures with environment snapshots), can involve reasoning over several parallel versions of code, many of which may have been long lost from the programmer's text editor due to destructive edits made in the service of subsequent redefinitions. In addition to reasoning over the hidden storage associated with code versions, the programmer can exploit knowledge of activation frames in the execution stack. In *The Bent Pyramid*, I manage to stuff away different versions of code I had evaluated and had them recalled at critical moments, without manual intervention, and without using any language features that would seem to have anything to do with storage and retrieval.

What else can we do with these ideas? Assuming that one could practice fluency with these hidden mechanisms to the same level of fluency exercised in *The Noodle*, I am unqualified to speculate on the range of what might be possible. It was the unsettling realization that a piece like *The Bent Pyramid* was possible in such a limited language that drove me to reevaluate my creation and what it revealed about the nature of livecoding.

5. CONCLUSION

It would seem that the combination of interactively redefinable functions and just-in-time evaluation of self-references (perhaps the essence of livecoding) are alone enough to crack open a new world of fractal complexity.

Livecoding is teeming with unfamiliar concepts that are just barely describable with our working vocabulary. I have spotted the ephemerality and simultaneous coexistence of multiple versions of algorithms; the overlaying of many phases of musical and software development into a single moment; the joint reasoning over when and how to update algorithms; a set of low-level machine operations that seem to have strange loops inherently threaded through them; and the concept of strategically managing code-at-hand (typed but not evaluated) as a live, performative gesture. I do not intend to have named or fully described every facet of the livecoder's conceptual landscape, but I hope to have indicated sufficient number of examples to warrant further inquiry (through methodical design as well as unguided exploration).

Perhaps, knowing of these mechanisms and understanding their potential for performativity will allow future livecoding language and system designers to surface the relevant computational details (potentially for the audience as well as the programmer) in a way that would make these uniquely-livecoding tricks more practical.

6. REFERENCES

[1] J. Cagan, *Engineering shape grammars: where we have been and where we are going*. New York, NY, USA: Cambridge University Press, 2001, pp. 65–92.

[2] N. Chomsky, "Three models for the description of language," *Information Theory, IRE Transactions on*, vol. 2, no. 3, pp. 113 –124, september 1956.

[3] M. H. Christensen. (2010) Structure synth. [Online]. Available: http://structuresynth.sourceforge.net/

[4] C. Coyne, M. Lentczner, and J. Horigan. (2012) Context free art: About. [Online]. Available: http://contextfreeart.org/mediawiki/index.php/Context_Free_Art:About

[5] B. Eno, "Generative music," *In Motion Magazine*, July 1996.

[6] D. Hofstadter, *I am a Strange Loop*. Basic Books, 2007.

[7] T. Magnusson, "Algorithms as scores: Coding live music," *Leonardo Music Journal*, vol. 21, pp. 19–23, 2011.

[8] M. Might. (2010) Continuations by example: Exceptions, time-traveling search, generators, threads, and coroutines. [Online]. Available: http://matt.might.net/articles/programming-with-continuations-exceptions-.../

[9] A. M. Smith. (2009) cfml.scm. [Online]. Available: https://github.com/rndmcnlly/cfml/blob/master/cfml.scm

[10] A. Sorensen, "Impromptu: An interactive programming environment for composition and performance," in *Proc. of the Australasian Computer Music Conference*, July 2009, pp. 2–4.

[11] G. Wang and P. R. Cook, "On-the-fly programming: Using code as an expressive musical instrument," in *Proc. of the Intl. Conf. on New Interfaces for Musical Expression (NIME'04)*, 2004, pp. 138–143.