# ‘My Little ChucKy’: Towards Live-coding with Grammatical Evolution
# Paper Type: Work In Progress

**Róisín Loughran** and **Michael O’Neill**
Natural Computing Research and Applications Group (NCRA)
University College Dublin, Ireland
roisin.loughran@ucd.ie

### Abstract

This paper proposes an initial framework for using PonyGE, a python implementation of Grammatical Evolution in generating ChucK files for music creation. We develop a number of grammars for creating individual instruments as shreds that can be added or sporked to the ChucK VM via the command line. A quicktime video example of the system running is provided. We propose that this strongly-timed system can be developed and generalised in the future to fully explore the capabilities of ChucK and to consider implications for Evolutionary Music.

## Introduction

Grammatical Evolution (GE) is a grammar-based Evolutionary Computation (EC) algorithm (Brabazon, O’Neill, and McGarraghy, 2015). Evolutionary systems, such as GE, develop solutions to a given problem by considering a population of individual solutions over a series of successive generations, rather than trying to deterministically improve one single solution. In recent years, such systems have been applied to creative tasks such as art and music. We present the initial framework for developing an evolutionary system that generates music using the live-coding language ChucK (Wang, Cook, and others, 2003). By the using the on-the-fly command programming capabilities of ChucK, shreds (each written as an individual file) can be added or *sporked* to a ChucK VM by the user at the command line. We develop a series of grammars that generate individual ChucK files for new shreds, that currently correspond to specific instruments. Each of these shreds can then be added or removed to the running virtual machine (VM) by the programmer.

‘My Little ChucKy’ is so named as it is an early stage system that uses PonyGE — a python implementation of GE in evolving the ChucK files. The following section introduces GE and live coding and describes some previous work in evolutionary composition. The remainder of the paper describes the system framework and proposes how it will be developed in future work.



## Background

This section introduces Grammatical Evolution, Evolutionary Composition and Live Coding for musical applications.

### Grammatical Evolution

GE is a grammar based algorithm based on Darwin’s theory of evolution. As with other evolutionary algorithms, the benefit of GE as a search process results from its operation on a population of solutions rather than a single solution. From an initial population of random genotypes, GE performs a series of operations such as selection, mutation and crossover over a number of generations to search for the optimal solution to a given problem. A grammar is used to map each genotype to a phenotype that can represent the problem under investigation. The success or ‘fitness’ of each individual can then be assessed as a measure of how well this phenotype solves the problem. Successful or highly fit individual reproduce and survive to successive generations while weaker individuals can be weaned out.

**Grammar** The creative capabilities of GE in part result from the choices offered within the mapping of the grammar. Typically, the genome is represented by a combination of 8 bit integers known as *codons*. These codons select the particular rule for a given expression according to the mod value from the number of choices for that rule.

$$\text{Rule} = (\text{Codon Integer Value}) mod (\#\text{ of choices}) \quad (1)$$

Using this we can introduce biases to our grammar by including multiple instances for preferred choices. For example, the operand depicted below offers three choices, two of which are choice1. Thus there is a 2:1 bias towards the selection of choice1 over choice2. Such biases can be exploited in the design of grammars.

```
<operand>:: = <choice1>|<choice1>|<choice2>
```

The grammar specified for the proposed system is strongly typed to produce legitimate ChucK code as detailed in the next section.

### Evolving Music

A number of previous studies have employed EC techniques for melodic composition. One of the most successful and well-known applications is GenJam (Biles, 1994) which

uses a Genetic Algorithm (GA) to evolve jazz solos. This system has been modified and developed into a real-time, MIDI-based, interactive improvisation system that is regularly used in live performances in mainstream venues (Biles, 2013). A modified GA was used in GeNotator (Thywissen, 1999) to manipulate a musical composition using a hierarchical grammar. Göksu et al evolved and evaluated both melody and rhythm separately using Multi-layerd Perceptrons (MLPs) (Göksu, Pigg, and Dixit, 2005). These evolved melodies were then mixed to produce verses and whole songs. Dahlstedt developed a system that implements recursively described binary trees as genetic representation for the evolution of musical scores. The recursive mechanism of this representation allowed the generation of expressive performances and gestures along with musical notation (Dahlstedt, 2007). Adapted GAs were used with local search methods to investigate human virtuosity in composing with unfigured bass (Munoz et al., 2016), with a grammar to augment live coding in creating music with Tidal (Hickinbotham and Stepney, 2016), and with non-dominated sorting in a multi-component generative music system that could generate chords, melodies and an accompaniment with two feasible-infeasible populations (Scirea et al., 2016).

**GE and Music Composition** While a number of systems used grammar based systems for music composition McCormack (1996), GE was first specifically used for this purpose in de la Puente, Alfonso, and Moreno (2002). In this paper GE generated melodies for a specific processor but the melodies produced were not presented or discussed. GE has been implemented for composing short melodies in Reddin, McDermott, and O'Neill (2009). From four experimental setups of varying fitness functions and grammars they determined that users preferred melodies created with a structured grammar. GE was again employed for musical composition using the Wii remote for a generative, virtual system entitled Jive, Shao et al. (2010). This system interactively modifies a combination of piece-wise linear sequences to create melodic pieces of musical interest.

GE was used with a number of different fitness measures in numerous versions of a system that created MIDI melodies by developing a grammar that expanded the genome into a series of notes, turns, chords or arpeggios. Early versions of the system used pre-defined fitness measures based on statistical tonal measures (Loughran, McDermott, and O'Neill, 2015b) or Zipfs Laws (Loughran, McDermott, and O'Neill, 2015a). Later version of the system were developed to emanate a self-adaptive system, whose fitness measure was based on the concept of conforming to the popular opinion of the population (Loughran and O'Neill, 2016). This resulted in a complex adaptive system that was self-referential and autonomous once it had been initialised. This system was generalised from a ranking-based system to a clustered based system in (Loughran and O'Neill, 2017).

## Live Coding

Live coding is a practice where software that creates music (and sometimes visuals) is written and manipulated in real-time as part of a live performance (Brown and Sorensen, 2009). Typically in a performance, the code is made visible on large screens, thus providing a more transparent experience to the audience. This leads to the possibility of glitches (or all-out crashes) within the software during the performance, the re-working or patching of which is part of the skill and nuance displayed by the composer (Collins et al., 2003). While a number of specific coding environments have been developed specifically for live coding music, live coding can be implemented in any computing language. The proposed system is developed using ChucK, a popular, strongly timed live coding programming language (Wang, Cook, and others, 2003).

# Method

This section offers an overview of the system, detailing the representation, grammar and fitness function used.

## ChucK

The ChucK usage in this version of our system is based on an adaptation of the on-the-fly programming synchronisation examples provided by the creators of ChucK, Perry Cook and Ge Wang, available at `http://chuck.cs.princeton.edu/doc/examples/`. These initial experiments involved the creation of individual grammars that could re-create a valid variation of each of the provided .ck files. GE was run independently multiple times, the run number specifying the grammar to be used and the name of the .ck file to be created. Each time the GE program is called, six individual .ck files are created. These can then be added or removed as ChucK shreds from VM by the programmer at the command line.

## Grammar

We have written six distinct grammars, each of which is run with GE to create a distinct instrument that can be sporked as a ChucK shred. In following the on-the-fly examples, each file begins with a specified code block followed by a `while` loop. Each grammar specifies the initial conditions for the given instrument and then introduces a flag before continuing to the code loop. Hence the first line in each grammar consists of:

```
<return> ::= <pre> L <code>
```

This states that the result will be comprised of some pre-code specified in `<pre>` followed by the body of code `<code>` and separated by the flag 'L'. The `<pre>` specifies the details for starting the code. For example in the first grammar, which creates the kick drum beat:

```
<pre> ::= .5::second => dur T;
T - (now % T) => now;
SndBuf buf => Gain g => dac;
me.dir() + "data/kick.wav" => buf.read;
.5 => g.gain;
```

This section of the grammar reads the sound file into the buffer, sets the gain and synchronises the resultant sounds to a period. This is currently hard-coded through the grammar — there are no non-terminals for GE to choose between. The remainder of the kick drum grammar consists of:

```
<code> ::= <line1> ; <line2> ; <line3> ;
<line1> ::= 0 => buf.pos
<line2> ::= <gain> =>buf.gain
<gain> ::= 0.8|0.82|0.84|0.86|0.88|0.9
<line3> ::= <dur>::T =>now
<dur> ::= 0.5 | 1 | 1
```

This grammar returns three lines of code. `<line1>` sets the play position to the beginning. `<line2>` offers a choice for the given gain and `<line3>` allows options for the duration, by advancing time. Note that the options for the `<gain>` are each equally likely, but there is a 2:1 bias towards advancing time by 1 rather than 0.5. Such biases can be introduced to the grammar as a design feature by the programmer.

The remaining grammars are implemented in a similar manner although some, such as that for snare-hop, allow multiple possibilities as to where in the period the instrument will sound:

```
<line1> ::= <gain> =>buf.gain
<gain> ::=  0.7 | 0.8 | 0.85  | 0.9
<line2> ::= where => buf.pos
<line3> ::= <single>|<single>|<mix>|<mix>|
   <mix>|<mix>|<double>
<single> ::= 2::T => now
<mix> ::= .25:: T => now;<line2>;.5::T => now;
   <line2>;1::T => now;<line2>; .25::T => now;
<double> ::= .75::T => now; <line2>;
   1.25::T => now
```

The above grammar offers three alternatives as to where (and how many times) the snare will sound within the bar `<single>`, `<mix>` and `<double>`.

The melodic content is created using Sin oscillators with a specified scale structure. For example, Grammar5 consists of:

```
<return> ::= <pre>L <code>
<pre> ::= .5::second => dur T;
T - (now % T) => now;
SinOsc s => dac;
.25 => s.gain;
<scale>@=> int scale[];
<scale> ::= [0,2,4,7,9]|[1,3,8,9,11]
  |[0,1,2,3,4]|[4,5,7,10,11]

<code> ::= <line1> ; <line2> ; <line3> ;
<line1> ::= scale[Math.random2(0,4)] =>
  float freq
<line2> ::= Std.mtof( 21.0 +
   (Math.random2(0,3)*12 + freq)) => s.freq
<line3> ::= <dur>::T =>now
<dur> ::= 0.25 | 0.5
```

The `<scale>` options in such a grammar dictates the degrees of the scale that are playable by the evolved instrument. A Sin oscillator based on multiples of `[0,2,4,7,9]` for instance would be pleasantly harmonic, whereas `[0,1,2,3,4]` would lead to more dissident intervals created by the instrument.

The above grammars are vey specific, and purposely limited to conform to the given example instruments. Allowing a choice of instruments — through an option in a .wav file chosen in the `<pre>` for example — would immediately allow more variation in the sounds produced by the resultant files. Such a change would require further efforts in designing the grammars to ensure that legal code is generated regardless as to which instrument was chosen.

### Fitness Measure

For this initial framework system we have implemented a purely random fitness function. While such measures may be alien to many EC researchers, as random fitness does not favour an individual for any justifiable reason, random fitness has been used in musical applications by incorporating only highly fit individuals within the population from initialisation (Biles, 2013; Eigenfeldt and Pasquier, 2012). From the grammar above, we only allow valid individuals for generating ChucK code. In the next version of the system, we plan to implement a more meaningful fitness measure such as one that responds to the evolution of the system itself, as proposed in Loughran and O'Neill (2016, 2017).

## Development

The code for this project is available at `https://github.com/roisis/PonyGE2/releases/tag/MuMe17`. Please note that this code is still in development. A quicktime video example of the system running can be found at `https://loughranroisin.wordpress.com/home/projects/`. This shows numerous runs of the GE program creating multiple versions of each ChucK Instrument that are added and deleted from the VM at the programmers whim. At the moment, we acknowledge that this system does not, on the surface, achieve substantially more than the code provided by Cook and Wang. However, we consider that the strength and potential in this system lies in the facility for expansion. The grammars specified above are currently limited to conform to the examples. The next step in our development is to to create more general grammars that can create valid and more expressive ChucK code. EC has a long history of use for program synthesis and is becoming an increasingly popular approach in Software Engineering (Harman, Mansouri, and Zhang, 2012). We plan to contribute to program synthesis in the live coding domain specifically using the ChucK environment.

As the grammars create code that is more generalised, the next most important step we envisage is in the creation of a more meaningful fitness measure. The current system only creates valid instruments, known in each run, so any individual is a valid selection. In EC, however, selection is what runs the evolution, and in EC on creative applications, such

as music production, this selection is based on a subjective measure — what makes one instrument or shred better than another? This point is far from trivial, and in effect causes the crux of the problem in applying EC to creative or subjective tasks. Once we have a more generalised system, we plan to address this issue in developing a more sophisticated fitness function. Ideally we would like to develop a fitness function that measures the worth of an individual, not through some pre-defined numerical measure, but rather from a response of the system to its current state such as that presented in Loughran and O'Neill (2016). Ideally this would lead to a self-adaptive system that could generate valid code for creating music with minimal human input.

Expansion in this way moves towards more autonomous music generation: We do not wish to write code that makes music, but to write code that writes code that makes music.

## Conclusion

We have presented an implemented and working framework for a system that augments live coding with GE, presently entitled 'My Little ChucKy'. While the system is still in development, at the time of writing we are currently and rapidly producing a more powerful and generalised system that will take advantage of the capabilities of both ChucK and GE. We plan to have a more sophisticated version of this system ready for presentation at MuMe 2017.

## Acknowledgments

This work is part of the App'Ed project funded by Science Foundation Ireland under grant 13/IA/1850.

## References

Biles, J. 1994. GenJam: A genetic algorithm for generating jazz solos. In *Proceedings of the International Computer Music Conference*, 131–131. International Computer Music Association.

Biles, J. A. 2013. Straight-ahead jazz with GenJam: A quick demonstration. In *MUME 2013 Workshop*.

Brabazon, A.; O'Neill, M.; and McGarraghy, S. 2015. Grammatical evolution. In *Natural Computing Algorithms*. Springer. 357–373.

Brown, A. R., and Sorensen, A. 2009. Interacting with generative music through live coding. *Contemporary Music Review* 28(1):17–29.

Collins, N.; McLean, A.; Rohrhuber, J.; and Ward, A. 2003. Live coding in laptop performance. *Organised sound* 8(3):321.

Dahlstedt, P. 2007. Autonomous evolution of complete piano pieces and performances. In *Proceedings of Music AL Workshop*. Citeseer.

de la Puente, A. O.; Alfonso, R. S.; and Moreno, M. A. 2002. Automatic composition of music by means of grammatical evolution. In *ACM SIGAPL APL Quote Quad*, volume 32, 148–155. ACM.

Eigenfeldt, A., and Pasquier, P. 2012. Populations of populations: composing with multiple evolutionary algorithms. In *Evolutionary and Biologically Inspired Music, Sound, Art and Design*. Springer. 72–83.

Göksu, H.; Pigg, P.; and Dixit, V. 2005. Music composition using genetic algorithms (GA) and multilayer perceptrons (MLP). In *Advances in Natural Computation*. Springer. 1242–1250.

Harman, M.; Mansouri, S. A.; and Zhang, Y. 2012. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR)* 45(1):11.

Hickinbotham, S., and Stepney, S. 2016. Augmenting live coding with evolved patterns. In *International Conference on Evolutionary and Biologically Inspired Music and Art*, 31–46. Springer.

Loughran, R., and O'Neill, M. 2016. The popular critic: Evolving melodies with popularity driven fitness. In *Musical Metacreation (MuMe), Paris*.

Loughran, R., and O'Neill, M. 2017. Clustering agents for the evolution of autonomous musical fitness. In *Evolutionary and biologically inspired music, sound, art and design.* Springer.

Loughran, R.; McDermott, J.; and O'Neill, M. 2015a. Grammatical evolution with zipf's law based fitness for melodic composition. In *Sound and Music Computing Conference, Maynooth*.

Loughran, R.; McDermott, J.; and O'Neill, M. 2015b. Tonality driven piano compositions with grammatical evolution. In *Evolutionary Computation (CEC), 2015 IEEE Congress on*, 2168–2175. IEEE.

McCormack, J. 1996. Grammar based music composition. *Complex systems* 96:321–336.

Munoz, E.; Cadenas, J.; Ong, Y. S.; and Acampora, G. 2016. Memetic music composition. *IEEE Transactions on Evolutionary Computation* 20(1).

Reddin, J.; McDermott, J.; and O'Neill, M. 2009. Elevated Pitch: Automated grammatical evolution of short compositions. In *Applications of Evolutionary Computing*. Springer. 579–584.

Scirea, M.; Togelius, J.; Eklund, P.; and Risi, S. 2016. Metacompose: A compositional evolutionary music composer. In *International Conference on Evolutionary and Biologically Inspired Music and Art*, 202–217. Springer.

Shao, J.; McDermott, J.; O'Neill, M.; and Brabazon, A. 2010. Jive: A generative, interactive, virtual, evolutionary music system. In *Applications of Evolutionary Computation*. Springer. 341–350.

Thywissen, K. 1999. GeNotator: an environment for exploring the application of evolutionary techniques in computer-assisted composition. *Organised Sound* 4(02):127–133.

Wang, G.; Cook, P. R.; et al. 2003. Chuck: A concurrent, on-the-fly, audio programming language. In *ICMC*.