Live Coding Poetry: The narrative of code in a hybrid musical/poetic context

ALEXANDROS DRYMONITIS

Independent Scholar, Greece
Email: alexdrymonitis@gmail.com

Live coding is a celebrated practice that is used in many areas, combined with a variety of artistic fields. Code poetry is a form of poetry with many variations, all of which have a common rule: the code that is or produces the poem must compile without errors. The meeting point of live coding and code poetry seems to have not yet been thoroughly explored, leaving space for experimentation and research. Certain attempts have already been made, where live coding is either approached through natural language or used to break up and merge chunks of existing poems, forming new ones. Computer code has also been used to write deterministic opera librettos, following the code poetry paradigm. This article focuses on the literary and artistic attributes of code, on code poetry and on the existing attempts to combine it with live coding. It also highlights the narrative attribute of musical live coding to formulate a rationale for combining live coding with code poetry in a musical context. The goal is to examine the possibilities of this combination, as well as how this can be achieved, from a technical point of view.

1. INTRODUCTION

Computer code has been approached from the perspective of poetry or literature in numerous ways by various researchers and artists. Even long before the computer, the combination of poetry with mathematics dates back to at least the medieval period (Toscano and Vaccaro 2020: 396). In 1960, the Oulipo group was founded, which focused on the application of mathematical structure to literature (ibid.: 394). Certain poems created by members of this group made use of 'high-level programming language commands', such as BEGIN, For, Do and Else (Forero 2021: 263). Graham compares hackers – essentially programmers – to painters, composers or novel writers (Graham 2004: 18). He takes inspiration into account as being necessary for writing software, and he even goes on to state that 'great software ... requires a fanatical devotion to beauty' (ibid.: 29), focusing on bad indentation or ugly variable names as a reference point to beauty – or ugliness for that matter. He states that computer programs 'should be written for people to read, and only incidentally for machines to execute', focusing on the literary nature of computer code, rather than its purely executional one.

From a different perspective, and in contrast to Graham, Cox, McLean and Ward suggest that the beauty of poetry, as with code, 'lies in its execution, and not simply its written form' (Cox, McLean and Ward 2000: 1). They also go on to compare code with poetry in a variety of ways, where their generative nature or the order of word placement are used as arguments. In their paper, they approach the relationship between code and poetry from the point of view of computer music code, something that can also be applied to live coding. Cox also writes that 'If program code is like speech inasmuch as it does what it says, then it can also be said to be like poetry inasmuch as it involves both written and spoken forms.' (Cox 2013: 17). Another mention of poetry through a computer music – and more specifically, a live coding – context is made by Rohrhuber, de Campo and Wieser, who anticipate 'a poetic language of code to find its way into programming and sound research' (Rohrhuber, de Campo and Wieser 2005: 4).

The various approaches to the relationship between computer programming and poetry might sometimes contradict each other. Nevertheless, a vivid interest in this interdisciplinary field exists, where researchers and practitioners attempt to highlight the common aspects of these two practices and the similarities of their nature. Still, none of these approaches seem to interpret computer code as poetry per se. An artistic practice that fills this gap is code poetry, but even though poems written in a programming language aim to create a functional program (Hopkins 1992: 391), in case the goal of this program is not to mutate the poem of the source code, this functionality is usually redundant. In this case, the focus shifts towards the poetry and away from the program or the algorithm.

Apart from poetry and the literary attributes of computer code, programming languages, but also programming itself, are being approached from different artistic perspectives, such as software art and conceptual software.[1] Even though these approaches do not immediately relate to poetry or musical live coding, they highlight a broader artistic attribute of code and

[1] https://runme.org/index.html; https://deprogramming.us/ (accessed 25 July 2023).

Organised Sound **28**(2): 241–252 © The Author(s), 2023. Published by Cambridge University Press. doi:10.1017/S1355771823000493

the act of programming. It is the similarities computer code bears to all these various art forms – which, even though subtle, are numerous – that drive my curiosity about how live coding and code poetry can be combined, and what is the potential of this combination.

Going back to live coding, the following questions arise:

1. (Can code poetry be coded live?
2. (If it can, what is the rationale for this?
3. (Can code poetry be combined with musical live coding?
4. (What is the rationale for the combination of music and poetry in a live coding context?

This article might not succeed in answering all these questions, though I will attempt to approach them from a critical point of view. Therefore, I will attempt to justify my rationale for this hybrid art form that seems to not have been thoroughly explored, and search for possible ways this can be achieved.

At this point, the reader might expect to read a definition of poetry, before this article proceeds. Throughout the history of poetry, many poets and scholars have either avoided defining poetry or have provided very different and even contradictory definitions. I will join the first strand and avoid attempting to define poetry, and why I consider my evolving approach to code poetry to be poetry. I will rest on applying Cage's approach to coining a new term for music: 'If this word "music" is sacred and reserved for eighteenth- and nineteenth-century instruments, we can substitute a more meaningful term: organization of sound' (Cage 1961: 3). In this context, if the word 'poetry' is sacred and reserved for poems with metre and rhyme, we can substitute a more suitable term: organisation of words.

The preceding statement can be interpreted as allowing every possible word structure to be considered to be poetry. In a live coding context – where a notion of a list of things that are *not* live coding is probably shorter than a list of things that are is common within the wider live coding community (Hutchins 2015: 147) – allowing such an open definition (or lack of definition) of poetry seems to agree with the philosophy of the live coding community. Such an open (un) definition of poetry can be further paralleled to Cage's notion of composition – or sound organisation – where 'the composer (organizer of sound) will be faced not only with the entire field of sound but also with the entire field of time' (Cage 1961: 5), as the potential poet (organiser of words) is faced with the entire field of vocabulary and the entire field of metre. A lack of both metre and rhyme though, where text is structured in paragraphs, can be considered to be prose, instead of poetry. In this context, by narrating a story with a structure of a programming language, I consider to be writing code poetry, as this structure, even if void of metre or rhyme, can resemble concrete poetry, and does not fit the structure of prose.

This article is centred around my existing work on live coding poetry, contextualised both by my rationale for this combination and the existing discourse on the artistic attributes of code, together with the limited literature and activity in musical live coding poetry. Section 2 is on related works, while sections 3–5 focus on code poetry, and the connection between code and concrete poetry and code and ASCII art, to highlight the artistic and poetic aspects of computer code. In section 6 I will discuss the narrative in musical live coding, something that connects to my rationale for live coding poetry in a sound-based performance context, which is developed in the section after that. In section 7 I will provide excerpts of my approach to live coding poetry from my past musical live coding poetry performances. Finally, in section 8 I will reflect on some of the intricacies of programming languages in the context of code poetry combined with computer music, and I will propose some approaches to how I believe the combination of musical live coding and code poetry can be achieved.

2. RELATED WORKS

This section discusses works that are relevant to the topic of live coding poetry in some way. This discussion aims to provide a foundation for the discourse that follows from section 5 onward by highlighting existing works whose creators have been occupied by the concept of the combination of live coding with (code) poetry.

CineVivo is a mini-language written in openFrameworks that enables the use of natural language to control the playback and application of effects to video files (Rodríguez, Betancur and Rodríguez 2019). It is not aimed at code poetry, but the use of natural language integrates narration into a live coding performance, outside of any technical-literary context. It works by replacing words in a custom parser, where the main command is `load` for loading videos. This word can be replaced by any other word, depending on the parser. The authors provide a few examples in their paper where the following line loads a video file named *bicycles* in layer *I*:

```
I love bicycles
```

Forero is attempting to live code poetry with an audiovisual result in his performance *Aimaako* (Forero 2018). In this performance, he concatenates numeric values to natural language phrases, which

https://doi.org/10.1017/S1355771823000493 Published online by Cambridge University Press

seem to control the audiovisual aspect in some manner. It is not clear how the natural language is treated by the system of the performance though. The following are a few lines copied from the documentation of this performance:

```
No_queia_ni_mirar_sus_cuerpos_llenos_de_rayas 5 5000;

Muy_pocos_hablan_su_idioma_y_nosotros_lo_vamos_olvidando 0 10000 0;

No_hubo_respeto 0;

Nadie_pregunto 1;
```

Both Forero's *Aimaako* and the CineVivo mini-language make use of natural language in a way that is not connected to most of the existing code poetry literature, as the typed poetry – or free text, in the case of CineVivo – is not written in a programming language, nor is it produced algorithmically by a program. I still consider these two examples to have relevance in the context of the combination of live coding with code poetry, as they attempt to convey the meaning of natural language through a live coding performance.

A work that relates to this article from a different perspective is *To code a dadaist poem* by Cotterill.[2] In this performance, Cotterill used live coding to split and re-combine a large bank of poetry in the public domain. Cotterill's description of his work states that he 'will … improvise live-coded sound using SuperCollider derived from the evolving poem, teasing out semantic and mimetic relationships between sound and text, and in turn adapting the sequencing and usage of poems according to the development of the music' (Cotterill 2015). This performance is a true example of live coding poetry, approaching the latter from the perspective of an algorithm producing a poem, rather than the source code being the actual poem.

Another attempt to combine live coding with code poetry is the opera *Echo and Narcissus* (Drymonitis and Manousakis 2022) by the artist group Medea Electronique. In this opera, the libretto[3] is written in pure Python, following the code poetry paradigm, and apart from expressing the story of the opera, it triggers all the sound processes, whether these are effects applied to the voices of the singers or audio generated by Unit Generators. All audio processes are written in Python with the Pyo DSP module (Bélanger 2016), but they are written in files that are being loaded in the main Python session where the libretto is being typed live during the performance. This breaks the convention of the actual source code being the poem, as a substantial body of code is used that is not shown to the audience, but, at that time, this approach was inevitable, due to time constraints and the extent of the knowledge I had in Python.

Another issue with this opera is that the libretto is deterministic and no variation of it is possible. This fact removes any aspect of improvisation, something that is an integral part of live coding. Nevertheless, I consider *Echo and Narcissus* to be closer to code poetry than the first two examples. This is because all the functional parts of the program are integrated into the code poem in a more seamless way than the second example, and because the first example, even though I consider it to be relevant in the context of live coding poetry, it is not aimed at poetry. I also consider *Echo and Narcissus* to provide a solid ground for research on how these two practices can be combined. An excerpt of the libretto is provided in section 7.

3. CODE POETRY

Code poetry is a form of poetry that utilises computer code in some way. From the existing literature on code poetry (Holden and Kerr 2016; Alvarez 2017: 35; Grillmair 2019: 15), it can be asserted that there are three main approaches to code poetry, which all share a common principle: the code used must compile without producing errors. One approach is to write a poem in a programming language instead of a natural one. Another approach is to write a poem in a programming language that, when compiled, results in a different poem. A third approach is to write an algorithm in computer code that will generate a poem when it is compiled. This third approach does not consider the source code as poetry per se. An example of a code poem that mutates when compiled by Alvarez (2017) is shown in Figures 1 and 2.

An example of generative poetry, which seems to fall in line with the code poetry approach where the code is not considered to be a poem but generates a poem, is the Book of All Words by Piwkowski where a computer program creates word strings consisting of one letter growing indefinitely where every word is similar to its previous one with only one letter being different (Kuchina 2018: 74). This work, part of which is shown in Figure 3, demonstrates the tautogram[4] principle, a poetical form broadly used by European poets (ibid.).

A good resource for code poetry is the./code –poetry website,[5] which includes 12 poems, each written in a different programming language (Holden and Kerr 2016). In this project, every poem creates some kind

[2]https://youtu.be/2dqd715LMQk (accessed 25 July 2023).

[3]A libretto, in an opera context, is the text of the opera that is sung by the singers.

[4]A text in which all words start with the same letter.

[5]https://code-poetry.com/ (accessed 25 July 2023).

https://doi.org/10.1017/S1355771823000493 Published online by Cambridge University Press

```
//: Playground - noun: a place where people can play

import UIKit

let (myWords, evolve) = ("mutate", "come alive")
let thisPoem = ["fracture poems", "and code"]
for words in thisPoem
{
    print(words + " into a new plane of existence")
}
let me = "introduce a world beyond"
print(" where words " + evolve)
```

Figure 1. Screenshot from Alvarez's article showing source code. Reproduced with the permission of the author.

```
fracture poems into a new plane of existence
and code into a new plane of existence
 where words come alive
```

Figure 2. Screenshot from Alvarez's article showing mutation. Reproduced with the permission of the author.

of ASCII art when it is compiled. Figure 4 shows the page of the poem written in Go, together with a snapshot of the ASCII art animation created by the poem. This animation could justify live coding such a poem, especially if the animation changes while the code is being written and executed. This justification also depends on whether the code poet is improvising, as this is an integral characteristic of the practice of live coding. Any of these two conditions would give a performative aspect to code poetry, answering the second question posed in section 1 of this article, on the rationale for live coding a code poem. Concerning the first question, whether code poetry can be coded live, the answer depends on whether the language used to write a code poem is a scripting programming language, therefore capable of being typed and executed live. These statements are personal assumptions based on my experiences with code poetry, as I have not yet encountered a live coding performance of code poetry that stands up to the improvisational fluency of musical live coding.

4. CODE AND POETRY BUT NOT CODE POETRY, AND OTHER (ART) FORMS

Computer code, in its various forms, seems to have more aspects in common with poetry – not code poetry – apart from its literary attributes. It also has common characteristics with other forms of art that incorporate computers in various ways, such as ASCII art.

4.1. Code and concrete poetry

Apart from the code poetry paradigm, computer code bears resemblances to concrete poetry, where the visual structure of a poem is more important, rather than the actual words (Hilder 2013). In Python, for example, indentation is very important for a program to run, as this language does not use curly braces to enclose code structures, but counts on correct indentation for the interpreter to be able to distinguish the various parts of a program. But in other languages that do use curly braces – or other means – to separate the various elements of a program, the visual form of the code is still considered important. Apart from Graham who emphasises indentation as an aspect of 'great' and beautiful software – as mentioned in section 1 – the International Obfuscated C Code Contest also highlights this importance, as one of its goals is 'to show the importance of programming style, in an ironic way' (Broukhis, Cooper and Noll 2020). In this context, computer code, without necessarily aiming at projecting its literary or poetic attributes, bears a resemblance to this form of poetry.

4.2. Brainfuck and ASCII art

A notable case of a programming language having common aspects with a form of art is that of brainfuck – intentional lowercase b (Chandra 2014: 119) – and ASCII art. Being an esoteric programming language, brainfuck aims mostly at challenging and amusing programmers. Its minimal set of commands makes programs written in this language very hard to decipher. For example, the following line is the famous 'Hello World' program written in brainfuck:

```
-[------->+<]>-.-[->++++++<]
>++.+++++++..+++.[->+<]>+++.-[-
>+++<]>.-[->+<]>-.+++.------.--------.
```

https://doi.org/10.1017/S1355771823000493 Published online by Cambridge University Press

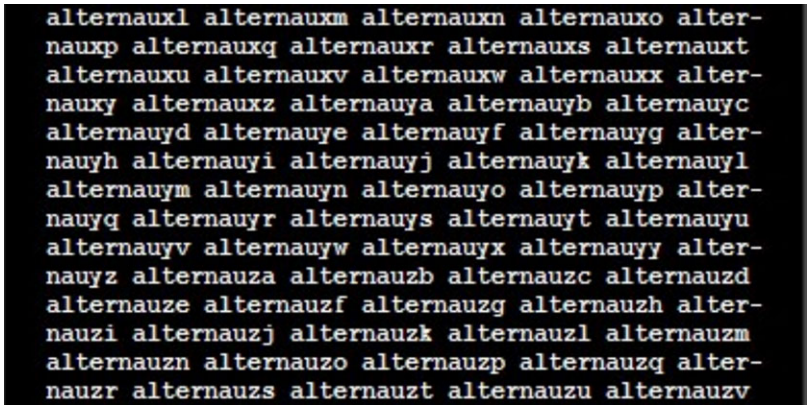

Figure 3. Screenshot from Kuchina's paper. Reproduced under a Creative Commons 4.0 licence.

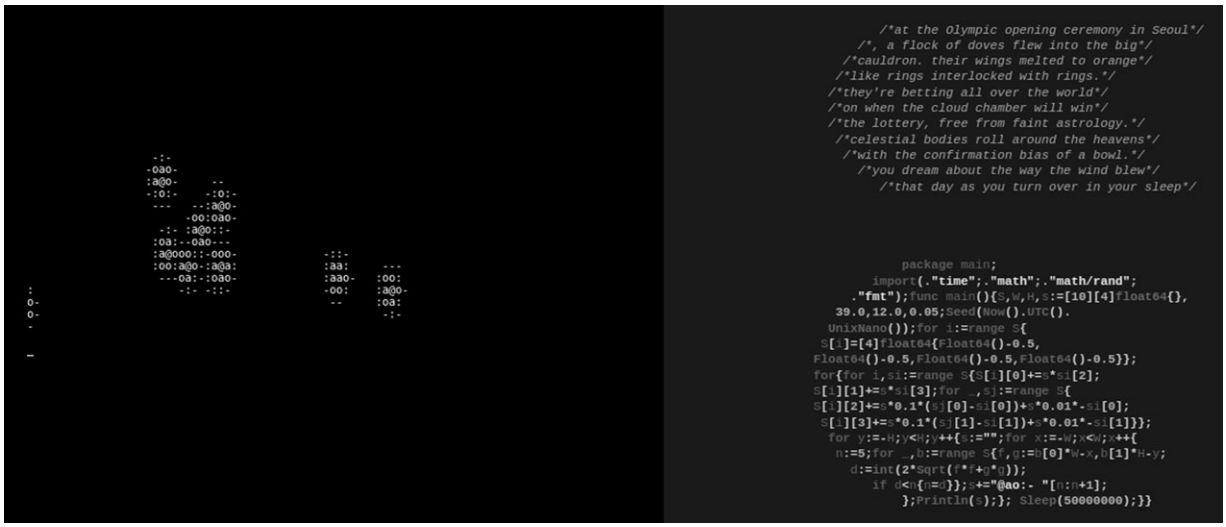

Figure 4. Screenshot from www.code-poetry.com website. Reproduced with permission of the website owners.

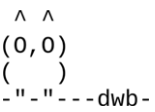

Figure 5. Owl sitting on a tree branch by Bake. Copied from the asciiart.eu website.

In my perception, this language bears a resemblance to ASCII art, where ASCII characters are used to depict an image, mimicking the brush strokes, pen lines and other painting techniques (O'Riordan 2002). For example, Figure 5,[6] is an example of ASCII art – the three letters at the bottom are apparently the artist's signature. If brainfuck used the characters of Figure 5 instead of the characters it uses, it could create ASCII art and at the same time a functional program that would produce output, even if that was only a 'Hello World' printed on the computer's console.

5. THE NARRATIVE OF LIVE CODING

Several programming languages created for live coding aim to simplify the creation of algorithms for the creation of musical patterns. Such languages include Tidal Cycles (McLean 2021), Ixi Lang (Magnusson 2011), FoxDot (Kirkbride 2021) and Sonic Pi (Aaron and Blackwell 2013). These languages use musical jargon for naming classes and methods to make the code easier to understand, from a musical point of view. As an example, the following two lines are copied from one of the tutorials on the official website of Tidal Cycles:[7]

[6]www.asciiart.eu/animals/birds-land. For terms of use see, www.asciiart.eu/terms-of-use (accessed 25 July 2023).

[7]https://tidalcycles.org/docs/patternlib/tutorials/course2 (accessed 25 July 2023).

https://doi.org/10.1017/S1355771823000493 Published online by Cambridge University Press

```
d1 $ n "'maj" # sound "supermandolin"
# legato 2 # gain 1.4
```

These two lines demonstrate an extensive use of musical jargon with names of acoustic instruments, and musical terms such as 'maj' for major and 'legato' being contained in this small code chunk. Ixi Lang names instruments following traditional music nomenclature, such as 'xylo' for xylophone, and FoxDot uses words such as 'pluck' for playing tones. Another example is the following line, taken from Sonic Pi's tutorials,[8] where electronic music jargon is used:

```
sample :loop_amen, attack: 1
```

Within a musical context, when live coding with one of these languages, there is a certain narrative that is being developed, where each line of code describes a certain musical action. The existence of a narrative can hold for coding in general, but the more low level a programming language is, the more this narrative can become esoteric and understandable only by experts – for example, extensive use of pointers in C or C++ can become very confusing to someone that is not familiar with this concept.

If we are to think outside of a programming frame and take into account a non-computer-literate audience, we can still consider the aforementioned live coding programming languages as narrative, always within a musical context. This means that a musical-literate audience will be able to follow the train of thought of the coder to a certain extent, even if they are not computer literate.

If we now think of live coding outside of a musical context – referring to the actual programming language, not the audio output – and still embrace a narrative that can be understood by a non-computer-literate audience, we are getting closer to a combination of live coding with code poetry. The question is: how can this be achieved? This question pertains to section 8, where various intricacies of programming languages within a code poetry context are discussed and two approaches to this combination are presented.

6. RATIONALE FOR LIVE CODING POETRY

In this section, I will attempt to answer the fourth question posed in section 1. The multitude of literature on live coding, including the International Conference on Live Coding,[9] the Live Coding book (Blackwell, Cocker, Cox, McLean and Magnusson 2022), and this volume of the *Organised Sound* journal, all witness a rich activity in the practice of live coding. Members of the live coding community, such as McLean, are characterised by a curiosity-driven activity that leads them to bring live coders closer to other practitioners (McLean 2015: 219). This fact, together with the notion of a variety of activities being considered as live coding (Hutchins 2015: 147), seems to provide an open frame where various fields of practice can find their place within live coding, whether these will be considered as live coding per se, or they will be combined with live coding in some way.

Code poetry is a practice that seems to have not yet been caught by the radar of the wider live coding community. Sure there are live coding works that incorporate code poetry in some way, something that has been demonstrated in the references section of this article. Additionally, the Live Coding book stretches the fields of practice where live coding happens (Blackwell et al. 2022: 25), and poetry is mentioned in various places. Its authors though, do not expand on actual code poetry through live coding. Combined with the little literature on live coding poetry, especially where the code is treated as poetry per se, we can assume that there is still little activity in this hybrid field.

As stated in section 1, my curiosity about how live coding can be combined with code poetry is driven by the similarities computer code bears to various artistic practices and its actual literary attributes that seem to occupy so many scholars. By combining these two practices, a new dimension is given to each, where live coding music expands its focus to poetry, and code poetry expands its focus to sound and music. Additionally, poetry, often relying on the recitation of the poem as a form of performance (Belle 2003), can be considered a performative form of art, in the same way that live coding is. The numerous examples of text-sound and sound poetry provided in Landy's *Compose Your Words* (Landy 2020), either of his own or of other artists' – such as Marinetti, the Oulipo group and Schwitters – highlight the performative character of poetry. Therefore, live coding and code poetry seem to have more characteristics in common, besides the actual code.

The expansion of dimensions in each practice that results from their combination poses certain challenges, mainly discussed in section 8. This combination also raises the following question: why should live coding be combined with code poetry? A simplistic, and perhaps naive answer, is that live coding artists can be characterised by curiosity, often attempting to combine live coding with other forms of expression (McLean 2015). If we focus on the performative and auditory aspects of poetry though, it seems that employing the practice of live coding in code poetry provides a medium that gives a performative and sonic character to the latter, a character

[8] https://sonic-pi.net/tutorial.html#section-2-1 (accessed 25 July 2023).
[9] https://iclc.toplap.org/ (accessed 25 July 2023).

https://doi.org/10.1017/S1355771823000493 Published online by Cambridge University Press

that is present in other forms of poetry as well. This combination opens new possibilities for live coding, as live coding poetry opens a portal for live coders to enter fields such as opera, recited/sound poetry, theatre, or any art form that relies on written or spoken text.

7. PERSONAL ATTEMPTS

In all my attempts to do live coding poetry that produces sound, I have used Python. But even with a language such as Python, writing code poetry live that can also produce sound is a challenging task. Up to now, I have been writing classes and functions with keywords I want to use in files I insert into the main Python instance. I then call them during the performance. These class methods and functions also trigger sound processes. My first attempt was a translation of three poems by Vakalopoulou from English to Python, where, apart from the poem, the code would also create sound. These were performed at the Music and Poetry Festival by the Music Is network in November 2015. The performance can be found online.[10] the following is the code poem *Ocean Floor Is Awake*:

```
>>> ocean_floor = _is._awake()
>>> one_eye = _is.made_of('pearl')
>>> ocean_floor.has(one_eye)
the other is waiting
>>>
>>> all = _is.set_up()
>>> all.for_us()
>>>
>>> all.to_meet_up('around_people')
>>>
>>> all.because('we_believed')
>>> all.people('in_public')
>>> all.would('forgive_us')
>>>
>>> all.our('words')
>>> all.our('thoughts')
>>> ocean_floor.our('passions')
```

This is a static poem that does not allow improvisation. In this attempt, I tried to escape Pythonic keywords such as 'is' by prepending an underscore. I also used a few strings and prepared a printed line: 'the other is waiting', shown without the Python interpreter prompt. The difficulty in this poem was that the translation had to be as close to the natural language as possible, and that alienated the code poetry from being code.

My second attempt was in collaboration with performer Catalano, during the Koumaria Residency by the artist group Medea Electronique, in 2016. I prompted Catalano to write a modular poem in English, which I then translated to Python. This second attempt provided more freedom, as I prepared the code so that certain functions could take more than one set of arguments. Each different argument would also provide a different sound. The result was a modular poem that could take a few possible forms. The following is an excerpt of the version of the performance of this poem, at the Onassis Cultural Centre, in Athens, in 2018, that can be found online:[11]

```
>>> sink.deep_in("the earth's shelter")
>>>
>>> free.the_exiled("light")
>>>
>>> Break.the_copper_made_birds()
>>>
>>> if disperse.closer_to("the earth"):
... release.the_exiled("sky")
...
>>> disperse.a_silent_window_to("the
light")
>>>
>>> sink.your_eyelids()
>>>
>>> release.the_exiled("night")
```

The approach to this modular poem was to create classes whose names make literate sense when combined with their methods. Also, some methods of some classes shared the same name with methods of other classes, as we see in the preceding example where `the_exiled` method name is shared between the classes `free` and `release`. Extensive use of strings was also applied in this poem, something that I attempted to avoid in my next attempt, the *Echo and Narcissus* libretto, which was written in collaboration with Poulou and Manousakis, both members of Medea. The following is an excerpt of the libretto, while the entire performance can be found online:[12]

[10]https://vimeo.com/146145222/da4d1d82d9 (accessed 25 July 2023).

[11]https://vimeo.com/187892132 (accessed 25 July 2023).

[12]https://youtu.be/1Btt4am2S2k (accessed 25 July 2023).

https://doi.org/10.1017/S1355771823000493 Published online by Cambridge University Press

```
>>> poet = Coder()
>>> rhapsode = Coder()
>>>
>>> myth = True
>>>
>>> with Words():
... poet.reconstructs(myth)
...
>>> poetry = 'rhythm' in 'words'
>>> poet.writes(' ' in 'python ')
>>> poetry = ' '
>>> python = poetry in 'code '
>>>
>>> nymphs = Nymphs()
```

The preceding libretto excerpt does not avoid strings altogether, but I tried to integrate them into the language specifics as much as possible. This is demonstrated where the `in` membership test operation checks if a white space string is included in the string `'python'`, or other places where `in` is used. This libretto is more Pythonic in its syntax, as I created variables of standard classes, such as Booleans, or classes I explicitly created in files imported to the Python session. I also used other keyboards, such as `with`, more frequently. A discussion on the music of this opera can be found in my PhD thesis (Drymonitis 2021: 40).

Recently I have attempted to write haikus without inserting pre-written files with classes and functions named after keywords that serve the poem. The Pyo module has to be imported though, and its audio server must be booted and started. Once these lines are written, a haiku can be written. The following are two haikus:

```
>>> give = Input(0)
>>> and_then = Chorus(give, depth=2)
>>> winter = and_then.out()

>>> winter = time.sleep(2)
>>> hear = PinkNoise().mix(2).out(1)
>>> summer = time.sleep(True)
```

Even though the number of syllables for a haiku is followed (the equals sign is not supposed to be pronounced), to make literate sense of these two haikus, the narration rhythm should split or concatenate certain lines, which breaks the haiku rule for the number of syllables per line. For example, the second haiku should be read as:

Winter time

Sleep to hear pink noise mix, to out one

Summertime, sleep true

In this attempt, I have utilised Pyo classes that fit the meaning of the poem, rather than writing a class of my own with a name that would serve my purposes. Both haikus produce audio and are functional in a Python interpreter with the Pyo module imported. The resulting sound though is very simplistic. Some variation could be given to the first haiku, as it takes audio input, applies a chorus effect and outputs it to the speakers. If the performer utilises natural feedback by placing the microphone close to the speakers, there will be some variation in the sound.

8. INTRICACIES AND POSSIBLE APPROACHES

In this section, I will discuss the intricacies of programming languages and possible approaches to combining live coding with code poetry. I consider the subject of this section to be highly subjective, therefore I will start by stating that what follows is my estimation of how the combination of live coding and code poetry can be achieved. Additionally, I have knowledge of a few programming languages only – these include C, C++, Python and a little bit of Lua – thus, my opinions are formed by a limited field of expertise.

8.1. Intricacies of programming languages in a code poetry context

The first aspect of code poetry I would like to examine is the punctuation marks and various symbols used in various programming languages, and how these are treated by code poets. Whether a programming language makes extensive use of curly or squared brackets, or various other symbols such as asterisks, slashes, ampersands, or others, these symbols are omnipresent and are not easily circumvented. Therefore, a code poem should take a stand on whether these symbols will be integrated into the poetry or ignored altogether. This decision though will most likely affect the destiny of the poem, as ignoring such symbols is easier to be realised in recited poetry rather than written. Nevertheless, there are examples of written poetry that attempt to ignore this special notation. *Ode to My Thesis*, a code poem written in Perl by Counterman (Hopkins 1992: 392, 397) has placed various punctuation marks to the far right of the screen, separating the text of the poem from these symbols. Still, these symbols are visible to the reader, even though the poem can be easily read without paying attention to them.

https://doi.org/10.1017/S1355771823000493 Published online by Cambridge University Press

Most Perl poems try to integrate the special symbols of this language by reading the $ sign as 'dollar' and @ as 'at' (ibid.: 392). To further complicate matters, if such symbols are to be integrated into the poem, it is not clear how this should be done. An example is a haiku by Wall where 'STDOUT' must be pronounced 'Standard Out', to fill in the necessary number of syllables of haiku (ibid.: 392).

Another aspect that affects code poetry is keywords. Most, if not all, programming languages have a set of keywords with a special meaning. For example, in C/C++/Python, the words 'for' or 'while' are keywords for creating loops. For a code poem to compile without errors, this means that such words have to be used within the scope of their special functionality, which means that their special role has to be integrated into the functional program of the poem, besides their literary attributes. Some languages such as Ruby and Lua conclude their function definitions and code structures created with 'for', 'while', or other keywords, with the keyword 'end'. This applies to nested structures, where the 'end' keyword will have to be repeated for every structure inside the nest. Unless we focus on concrete or visual poetry, unavoidably, this affects any poem written in these languages. Haskell on the other hand, makes extensive use of the keywords 'let' and 'where'. For a single poem, these words could be nicely integrated, but for extensive use for writing code poems, using these words repeatedly could result in predictability.

Perl is a language that, even though its keywords are in the magnitude of over two hundred, appears to provide a certain amount of freedom, as it does not complain when many of these keywords are being abused (Hopkins 1992: 392). On the other hand, Perl uses an extensive set of symbols, something that brings us to the first point of this section. There are a few languages that make a rather limited use of punctuation marks and special symbols, including Ruby and Lua, but also Python. Ruby and Lua bare the obligation of the 'end' keyword, as stated earlier, but Python is based on indentation only to conclude its function definitions and other code structures. Additionally, Python replaces many symbols used in other languages with English words. For example, the Boolean AND is expressed with the English word 'and' in Python, whereas in C/C++ it is expressed with a double ampersand symbol. The same applies to other Boolean operators. By escaping punctuation though, we fall to the second point of this section, which is keywords. It seems that within the programming language domain, it is impossible to avoid both. If it were possible, we would be probably talking about natural languages instead. And, in my perception, this is the challenge code poets are called to overcome, but – inspired by Stravinsky – a challenge that feeds inspiration rather than limits creativity (Stravinsky 1970: 65).

The last element of the combination of live coding with code poetry that we need to examine is the audio capabilities of a programming language or framework. From the multitude of programming environments for audio that exists, the literary attributes of the language of each environment should be taken into consideration. A DSP framework with potential literary attributes is Python with the Pyo module. Even though Pyo is not widely used, it is a very efficient module for DSP written in C with its interface in pure Python. Bearing in mind the advantages Python can have over other languages in a code poetry context, this framework seems to provide the desired potential for live coding poetry. The sc3 module for Python that controls SuperCollider's scsynth (Samaruga and Riera 2022) can also be considered, as it is a fully featured DSP module for Python. At the time of writing though, this module is still in beta with several functionalities still missing and its documentation under development (Samaruga, Silvani and Saladino 2021).

8.2. Possible approaches to combining live coding with code poetry

This subsection provides a conceptual framework on how live coding and code poetry can be combined in a single practice by discussing two possible approaches. It also attempts to answer the third question of section 1. The discussion in this subsection though is more theoretical than practical. The reader is thus invited to take the following information with a critical point of view.

If we suppose that we have solved the language issue in a poetry and an audio context, separately, we should now consider how to merge the audio and poetic attributes of the language we can use. If one is to use an existing programming language intact, one is restricted to the names of classes that produce or process audio, as these classes must be called and eventually integrated into the poem. The equals sign together with other punctuation marks such as squared or round brackets, even if sparse, will be still present. If the live coding convention of 'show us your screens' is to be followed, then these punctuation marks will inevitably make their way into the poem, so they must somehow be integrated as well. To overcome these issues we have to enable a somewhat free vocabulary with a strict syntax – to justify the word 'code' in the term 'code poetry' – that would bind the musical output to the poetic phraseology.

At the time of writing, Artificial Intelligence (AI)[13] is a very popular practice, and this is evident through

[13]AI is a practice where an algorithm is tuned based on data, a process called training, and provides output on unseen data, based on its training.

https://doi.org/10.1017/S1355771823000493 Published online by Cambridge University Press

the volume of research and literature on this field. In this context, even though I have not yet tested it in the context of live coding poetry, AI could be deployed in a way that will enable the generation of audio, or the control of synthesis parameters, based on the written text. The practice of sentiment analysis (Bhaumik and Yadav 2021: 59) – which provides values for the sentiment of text, whether that is positive, negative, or neutral – can prove helpful, as it can serve as the input to AI, based on which the latter will produce or control sound. To achieve this, if Python is to be used, a custom Python interpreter is necessary, as undefined variable names are bound to raise an error with Python's default interpreter. A custom interpreter will enable a more free vocabulary. As an example, the following line, from the *Echo and Narcissus* libretto, which reads:

```
I = Narcissus()
```

could be changed to:

```
I.am(Narcissus)
```

Thus, we can avoid the equal sign and render the line more readable and easier to include in a live coding session. The first case not only includes the equal sign – which can of course be included in the poem, but its pronunciation is ambiguous, as it can be read both as 'I equal Narcissus' or 'I am Narcissus' – but it also needs the class 'Narcissus' to have been defined before it is invoked. That was the main issue with *Echo and Narcissus*. In the second case, the custom interpreter will not complain about the undefined words – all three words, in this case – and no class 'Narcissus' is necessary to have been defined.

Concerning the custom interpreter and the preceding example, two approaches are possible. One is to convert the line to a comment, so the interpreter will ignore it and use it only in the context of AI. This approach seems more suitable if AI is used to generate audio, instead of controlling synthesis parameters – which is the second approach. This approach is also more involved than the second one, because AI models that generate audio need many hours of audio recordings as training data, and can be very CPU expensive both during their training and when used for audio generation.

The second approach is to iterate through this line until all undefined variables have classes or values assigned to them. The interpreter should probably be aware of the fact that 'I' must become an object of a class, 'am' a method of that class, and finally, 'Narcissus' should be mapped to another class that will be passed as an argument to the method the word 'am' will be mapped to. The following lines are an example of how this mechanism could map the line given earlier:

```
I = SuperSaw()
Narcissus = Sine(freq=185, mul=200, add=200)
I.setFreq(Narcissus)
```

This way, 'I' will become an object of the `SuperSaw` class, 'Narcissus' an object of the `Sine` class, and 'am' will be mapped to the `setFreq` method, which is common between many oscillator classes in Pyo. The preceding example will result in a frequency modulation of a super saw oscillator modulated by a sine wave oscillator. The last line is now very similar to the original, with the replacement of 'am' with 'setFreq' being the only difference.

The classes that will be chosen in the preceding process are the responsibility of the AI. The training of the AI should be undertaken by each user separately for the system to represent the audio aesthetics and the desired connections between text and sound of the respective user.

The following lines below, inspired by the libretto of *Echo and Narcissus*, are a similar example:

```
echo babbles(randomly):
Hera.is_distracted(intentionally)
```

The preceding pseudo-code has the structure of a function definition in Python. Thus, with the proposed approach, it can be replaced with the following code:

```
randomly = random.randrage(50, 200)
Hera = SineLoop(freq=200, feedback=0.05)
def babbles(randomly):
intentionally = randomly
Hera.setFreq(intentionally)
```

After these mappings are made, 'Hera' will be assigned to an object of the `SineLoop` class, and 'babbles' will become a function that will set a random frequency to the 'Hera' object. In the myth of 'Echo and Narcissus', Echo was a Nymph who intentionally distracted Hera with her babble so that Zeus could flirt with other women. Thus, the mapping of the pseudo-code of this example can depict part of the myth through sound, alongside the actual code poetry.

A programming environment with a free vocabulary but a strict syntax can provide a framework for expression that is free to a certain extent, in the domain of code poetry, combined with sound. Python includes modules for sentiment analysis and a big variety of machine learning models. It also provides the capability to create a custom interpreter. All this, combined with the Pyo module, provides all the necessary tools for the creation of the proposed approaches to live coding poetry, all in the same programming language.

https://doi.org/10.1017/S1355771823000493 Published online by Cambridge University Press

9. CONCLUSIONS

This article has attempted to contribute to the discourse and the literature on the combination of live coding and code poetry, as well as to provide a critical point of view on how these two practices can be combined. Even though the focus of this article has been the integration of code poetry in live coding, or how to write code poetry live, hence do live coding poetry, the intention has been to combine poetry with music or sound, as a way to provide another dimension both to code poetry and to musical live coding. Expanding on the actual music that can be produced by this combination is beyond the scope of this article, as its potential length would outgrow the available article length of the *Organised Sound* journal. Nevertheless, links to existing works, or references to literature that discusses this, are provided in certain places in this article.

By acknowledging the small volume of literature and the little activity on this hybrid art form, together with the broad perception of what is, or what can be live coding, and the curiosity that characterises the live coding community, this article aims to initiate or continue the discourse on live coding poetry. I believe that this combination has not been explored thoroughly. On the other hand, with live coding approaching many art disciplines, and with existing software and programming languages providing the necessary tools to realise this combination, I believe that the conditions are ripe for diving deeper into this hybrid art form. Such a combination can open up new fields of practice for live coding, something that seems to be always welcomed by its community.

This combination though projects several challenges. These are connected to the intricacies of programming languages in a literary context, but also to the obligatory use of names of classes that produce or process audio, in a poetic context. A proposed approach to overcome these challenges is to create an environment for live coding poetry that will be tolerant towards a more free vocabulary, but strict with syntax. This environment could make use of sentiment analysis to derive the poetic intention of the coder, and AI to invoke audio classes or to generate audio, based on this analysis.

From the multitude of programming languages that exist, Python seems to be a choice that provides the necessary utilities for the creation of such an environment, as its repositories include modules for sentiment analysis and AI, besides DSP and the ability to create a custom interpreter. The latter is necessary to liberate the vocabulary of the language and enable a more free expression, as undefined variables will internally be assigned to objects or values, or converted to comments. Besides these attributes, Python makes extensive use of English words and, compared with other languages, limited use of punctuation marks, a fact that serves code poetry well.

Acknowledgements

I would like to thank Christian Iñigo D. L. Alvarez, Daniel Holden and Chris Kerr for giving me permission to use screenshots from their articles or websites in Figures 1, 2 and 4. Figure 3 is published under a Creative Commons Attribution 4.0 International license (CC BY 4.0), which can be found at https://creativecommons.org/licenses/by/4.0/. Figure 5 is published under the terms found in www.asciiart.eu/terms-of-use.

REFERENCES

Aaron, S. and Blackwell, A.F. 2013. From Sonic Pi to Overtone: Creative Musical Experiences with Domain-Specific and Functional Languages. *Proceedings of the First ACM SIGPLAN Workshop on Functional Art, Music, Modelling & Design*. New York: Association for Computing Machinery, 35–46. https://doi.org/10.1145/2505341.2505346.

Alvarez, C. I. D. L. 2017. The Intrinsic Mutability of Code Poetry Uncovers New Notions of Poetic Design. *Philippine Humanities Review* **19**(1). https://journals.upd.edu.ph/index.php/phr/issue/view/630.

Bélanger, O. 2016. Pyo, the Python DSP Toolbox. *Proceedings of the 24th ACM International Conference on Multimedia*. New York: Association for Computing Machinery, 1214–17. https://doi.org10.1145/2964284.2973804.

Belle, F. 2003. The Poem Performed. *Oral Tradition* **18**(1), 14–15. https://doi.org/10.1353/ort.2004.0007.

Bhaumik, U. and Yadav, D. K. 2021. Sentiment Analysis Using Twitter. In *Computational Intelligence and Machine Learning. Advances in Intelligent Systems and Computing*. Singapore: Springer, 55–66. https://doi.org/10.1007/978-981-15-8610-1_7.

Blackwell, A. F., Cocker, E., Cox, G., McLean, A. and Magnusson, T. 2022. *Live Coding: A User's Manual*. Cambridge, MA: MIT Press.

Broukhis, L., Cooper, S. and Noll, L. 2020. The International Obfuscated C Code Contest. www.ioccc.org/ (accessed 25 July 2023).

Cage, J. 1961. *Silence*. Middletown, CT: Wesleayan University Press.

Chandra, V. 2014. *Geek Sublime*. Minneapolis, MN: Greywolf Press.

Cotterill, S. 2015. ICLC Performances. https://iclc.toplap.org/2015/performances.html (accessed 25 July 2023).

Cox, G. 2013. *Speaking Code*. Cambridge, MA: MIT Press.

Cox, G., McLean, A. and Ward, A. 2000. The Aesthetics of Generative Code. *Proceedings of the International Conference on Generative Art*, Rome.

Drymonitis, A. 2021. The Artists Who Say Ni!: Incorporating the Python Programming Language into Creative Coding for the Realisation of Musical Works.

PhD thesis, Birmingham City University. https://doi.org/10.13140/RG.2.2.27923.55841.

Drymonitis, A. and Manousakis, M. 2022. Echo and Narcissus: Live Coding and Code Poetry in the Opera. *Proceedings of the International Computer Music Conference, ICMC*, Limerick, Ireland.

Forero, J. 2021. Code, Poetry and Freedom. *Proceedings of the 9th Conference on Computation, Communication, Aesthetics & X*, 261–77.

Graham, P. 2004. *Hackers and Painters: Big Ideas from the Computer Age*. Sebastopol, CA: O'Reilly.

Grillmair, R. M. 2019. Code and Poetry An Exploration of Logic throughout Art, Computation and Philosophy. Master's thesis, University of Arts, Linz, Austria.

Hilder, J. 2013. Concrete Poetry and Conceptual Art: A Misunderstanding. *Contemporary Literature* **54**(3): 578–614. https://doi.org/10.1353/cli.2013.0034.

Holden, D. and Kerr, C. 2016. ./code –poetry. https://code-poetry.com/ (accessed 25 July 2023).

Hopkins, S. 1992. Camels and Needles: Computer Poetry Meets the Perl Programming Language. *Proceedings of the USENIX Winter 1992 Technical Conference*. San Francisco, 391–404.

Hutchins, C. C. 2015. Live Patch/Live Code. *Proceedings of the First International Conference on Live Coding*, Leeds, 147–51. https://doi.org/10.5281/zenodo.19346.

Kirkbride, R. 2021. FoxDot. https://foxdot.org/ (accessed 25 July 2023).

Kuchina, S. 2018. On Generative Poetry: Structural, Stylistic and Lexical Features. *Matlit.* **6**(8): 73–83. https://doi.org/10.14195/2182-8830_6-1_5.

Landy, L. 2020. *Compose Your Words.* Philadelphia, PA: Intelligent Arts.

Magnusson, T. 2011. The IXI Lang: A SuperCollider Parasite for Live Coding. *Proceedings of the International Computer Music Conference, ICMC*, Huddersfield, UK.

McLean, A. 2015. Reflections on Live Coding Collaboration. *Proceedings of the Conference on Computation, Communication, Aesthetics & X, xCoAx*, Glasgow, 214–20.

McLean, A. 2021. Tidal Cycles. https://tidalcycles.org/docs/ (accessed 25 July 2023).

O'Riordan, K. 2002. ASCII Art. In S. Jones (ed.) *Encyclopedia of New Media*. Chicago: University of Illinois at Chicago, 15–16.

Rodríguez, J., Betancur, E. and Rodríguez, R. 2019. CineVivo: Livecoding Language for Visuals. *Proceedings of the Sixth International Conference on Live Coding, ICLC 2019*, Madrid, Spain.

Rohrhuber, J., de Campo, A. and Wieser, R. 2005. Notes for Language Design for Just in Time Programming. *Proceedings of the International Computer Music Conference*, Barcelona, Spain.

Samaruga, L. and Riera, P. 2022. A Port of the SuperCollider's Class Library to Python. *Proceedings of the 17th International Audio Mostly Conference (AM '22)*. New York: Association for Computing Machinery, 137–42. https://doi.org/10.1145/3561212.3561250.

Samaruga, L., Silvani, D. and Saladino, I. 2021. SuperCollider library for Python. https://github.com/smrg-lm/sc3 (accessed 25 July 2023).

Stravinsky, I. 1970. *Poetics of Music in the Form of Six Lessons*. Cambridge, MA: Harvard University Press.

Toscano, E. and Vaccaro, M. A. 2020. François Le Lionnais and the Oulipo. In M. Emmer and M. Abate (eds.) *Imagine Math 7*. Cham: Springer. https://doi.org/10.1007/978-3-030-42653-8_23.

VIDEOGRAPHY

Forero, J. 2018. Aimaako. *YouTube*. https://youtu.be/w_t-gm8mXAM (accessed 25 July 2023).