Compositional Computational Constructive Critique

Or, How My Computer Learned to Appreciate Poetry

Jennifer Hackett
University of Nottingham
jennifer.hackett@nottingham.ac.uk

Abstract

A well-typed work of art should not sound wrong:
we can use DSLs to get this boon.
Existing work has cover'd this for song,
at least so far as harmony and tune.
Well, verse contrains the author's pencil thus:
In English verse, the syllables adhere
to certain contours. Catenation plus
the empty form create a monoid here.
But not all poems stay within their meter;
some, like this, can be a little free
and so our monoid has to let us teeter
on the edge, as tunes oft do with key.
Our monoid must be *fuzzy* at its heart
to let us teach computers of this art.

ACM Reference format:
Jennifer Hackett. 2018. Compositional Computational Constructive Critique. In *Proceedings of the ACM SIGPLAN International Workshop on Functional Programming in Art, Music, Modelling and Design, St Louis, MO, USA, 29 Sept 2018 (FARM),* 7 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Computer creativity is a deep problem that brings together many disparate fields. Computer *checking*, however, is much simpler: computers have been checking the creations of humans for almost as long as they have existed [IBM Programming Research Group 1954]. While these checks can sometimes be aggravating, they can also help a human creator to develop something better than they could have otherwise hoped to create [Brady 2017]. At its best, a computer checker can play the role of an assistant or a teacher, pointing out where your work can be improved and possibly even suggesting ways to do so.

FARM, 29 Sept 2018, St Louis, MO, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Most work on computer checking focuses on fairly binary or ternary situations: a thing is either good, questionable or wrong. A compiler contains a set of rules that spell out exactly what makes a valid program, and given an input program it can check it rigorously against these rules, telling the user if it fits the rules perfectly (by accepting the program), if it does something potentially problematic but not exactly *wrong* (by emitting a warning) or if it breaks the rules outright (by emitting an error). In this way, a compiler will verify many important properties of the code before the program is executed, leading to the oft-quoted slogan "a well-typed program does not go wrong".

This is all well and good for tasks where the rules are rather rigid, like programming and mathematical proofs, but for many tasks "goodness" is a much more fuzzy notion. This is especially the case in artistic pursuits, where rules are often deliberately bent for artistic effect. There are many rules and guidelines associated with Bach's chorales — so much so that imitating his style is a common task given to students of music [Pankhurst 2009] — but even Bach himself was known to deviate from the formula. In order to check these kinds of creations, we need to embed the fuzziness into the computer checker itself, so as to allow for artistic license. Furthermore, this checking should be *compositional*, allowing the system to provide constructive criticism by suggesting which parts of the work contribute the most to a poor score.

This paper aims to address this problem of fuzzy checking for *poetic meter*, the pattern of stresses found in a line of poetry. Given a line of text and a poetic form, we would like a computer program to tell us how well the line fits the expected meter and which words in the line could be changed to improve it. The key insight that allows for both compositionality and fuzziness is that poetic meter can be treated as a *monoid*, resulting in a simple DSL for specifying meter. I have implemented this idea in a Haskell library, `hpoem`, available online at https://bitbucket.org/jenhackett/hpoem/. The structure of this paper roughly follows the process of developing the library, giving a (slightly simplified) account of the iterative design process.

Some details of Haskell that obscure the general ideas have been elided. In particular, I give type class instances for type synonyms *without* wrapping the data in `newtype`s.

2 Related Work

This work was directly inspired by the work of Szamozvancev and Gale [2017], who provide a Haskell DSL for music where "well-typed-music does not sound wrong". Quick and Hudak [2013] present a grammar-based approach to automatic music *composition*, also in Haskell, using a probabilistic approach to account for the fact that some productions are more common than others. However, this strand of work is not limited to the world of functional programming: Huang and Chew [2005] present a grammar checker for contrapuntal compositions written in Java. Music is particularly well-suited to this kind of approach, especially classical works where the rules are rather rigid. For linguistic pursuits, there are automatic checkers for simplified English grammar [Adriaens and Schreors 1992; Hoard et al. 1992], which are chiefly used to ensure the broad readability of technical manuals.

There is a fair amount of work on computer analysis of artistic pursuits. Kao and Jurafsky [2012] give a computational analysis of poetry, comparing the writings of amateur poets with award-winning poets to discover the features that make a poem "beautiful". Simonton [1986] applies a similar approach to music, analysing what features make a musical composition successful. Plamondon [2006] discusses how computers can be used to analyse poems to discover their meter and rhyme scheme. This is in some sense dual to my own work: whereas I aim to work out how well a poem fits a specific meter, Plamondon aims to guess the meter starting from the assumption that the poem is a good example of it.

There are a number of interesting directions in human-computer artistic collaboration. Yan et al. [2013] treat poetry as an optimisation problem, starting with intent from the user and then trying to build a poem that fits with the intent and the rules of the form. Botnik Studios [2016] provide *Voicebox*, an online tool for writers that suggests possible next words for a text based on a corpus from a specified source; this can be thought of as a Markov-chain approach with an added degree of interaction. Finally, there is the *Pentametron* [Bhatnagar 2012], which analyses tweets for poetic meter in order to create surrealistic "found poems".

3 Background

3.1 Accentual-Syllabic Verse

Poetry is a written or spoken art form that juxtaposes the *literal* meaning of words with their aesthetic qualities. English speakers often focus on *rhyme* as the main distinguishing feature of poetry from prose, where the ends of words or lines share some similarity in sound, but there are other factors of equal or greater importance. Old English poetry is primarily alliterative, where the *beginnings* of words share similar sounds. Japanese poetry is syllabic: a *haiku*, for example, consists of 17 *morae* (a unit akin to the English syllable) in groups of five, seven and five. Modern English poetry is a form of *accentual-syllabic* verse, based around the notion of *meter*, albeit with a significant minority[1] of *free verse* which eschews a rigid structure.

In accentual-syllabic verse, the words are expected to follow a particular rhythmic pattern. A line is divided into metrical *feet*, which can be thought of as like beats; each foot consists of a number of syllables with a fixed stress pattern. If a line fits the expected pattern of metrical feet, we say that it *scans*. The chart below lists a number of different types of metrical foot, with — representing a stressed syllable and ⏑ representing an unstressed syllable.

Name	Syllable structure	Example words
trochee	— ⏑	auntie towel farthing counter
iamb	⏑ —	upon until repel defend
dactyl	— ⏑ ⏑	omnibus albatross thundercloud infinite
amphibrach	⏑ — ⏑	avowal descending intended confounded
anapæst	⏑ ⏑ —	understand indiscreet Tenerife undercut (as a verb)

Many meters are specified simply by choosing a particular type of foot and a number of times that foot occurs per line. For example, the phrase "Teenage Mutant Ninja Turtles" [Eastman and Laird 1984] consists of four *trochees*, forming a line of *trochaic tetrameter*. Compare this to the phrase "Two households, both alike in dignity" [Shakespeare 1599], which contains five *iambs*[2] forming a line of *iambic pentameter*. A poetic form, in turn, is specified by choosing a number of lines and a meter for each line, and possibly a rhyme scheme. For example, a sonnet is formed from fourteen lines of iambic pentameter, while a limerick is formed from two lines consisting of three feet each (usually anapæsts or amphibrachs), another two lines consisting of two feet each, and a final line of three feet. Here are examples of each of these forms:

[1] Free verse is clearly in the minority if you count rap as a form of poetry, which I do. Discounting rap, free verse is likely in the majority. Readers who think it absurd to count a form abundant with sexual innuendos and diss-tracks as poetry are referred to the works of Catullus [1472].

[2] Arguably, the first iamb is actually a spondee, consisting of two stressed syllables.

> Let me not to the marriage of true minds
> Admit impediments. Love is not love
> Which alters when it alteration finds,
> Or bends with the remover to remove:
> O, no! it is an ever-fixed mark,
> That looks on tempests and is never shaken;
> It is the star to every wandering bark,
> Whose worth's unknown, although his height be taken.
> Love's not Time's fool, though rosy lips and cheeks
> Within his bending sickle's compass come;
> Love alters not with his brief hours and weeks,
> But bears it out even to the edge of doom.
> If this be error and upon me proved,
> I never writ, nor no man ever loved.
>
> Sonnet 116 — Shakespeare [1609]

> Upon high Olympus, great Zeus
> Muttered angrily, "Oh, what the deuce!
> It takes spiced ambrosia
> To get the nymphs cosier
> And Hera supplies grapefruit juice."
>
> She's No Dope — Asimov [1981]

However, meter is rarely adhered to completely, and neither of the above poems adheres strictly to its poetic form. Sonnet 116 begins with two trochees, rather than two iambs:

— ◡ — ◡ ◡ — ◡ ◡ — —
Let me not to the marriage of true minds

This is known as "trochaic substitution". There are also lines with a so-called "feminine" ending, where an extra unstressed syllable appears at the end:

◡ — ◡ — ◡ — ◡ — ◡ – ◡
Whose worth's unknown, although his height be taken.

Similarly, the limerick diverges from the strict meter, with the second line containing an extra unstressed syllable at the start as compared to the first and final lines that it mirrors:

◡ — ◡ ◡ — ◡ ◡ —
Upon high Olympus, great Zeus
◡ ◡ — ◡ ◡ — ◡ ◡ —
Muttered angrily, "Oh, what the deuce!"

Rather than being completely prescriptive, the meter provides a loose blueprint for the text to follow. In other words, poetry is not strongly-typed!

3.2 Meter as a Monoid

Let's do some maths. A *monoid* is a set M equipped with an operation $- \cdot - : M \times M \to M$ and neutral element $e : M$ that satisfy the following laws:

$$l \cdot (m \cdot n) = (l \cdot m) \cdot n$$
$$e \cdot m = m$$
$$m \cdot e = m$$

For example:

- The set of whole numbers $\mathbb{Z}$ with the operation + and neutral element 0
- The set of lists with the operation of contatenation ++ and neutral element the empty list []
- The set of truth values $\{true, false\}$ with the operation "and" and neutral element *true*

In the programming language Haskell, monoids are represented by a *type class*, a way to associate operations with a particular type:

```
class Monoid m where
  mempty  :: m
  mappend :: m -> m -> m
```

Here, `mappend` plays the role of the monoid operation $\cdot$, while `mempty` plays the role of the neutral element e. Essentially, this code tells the compiler that you want to use different versions of `mempty` and `mappend` for different data types, and that it should work out from context which one you mean.

Why mention monoids? Well, because poetic meter *is* a monoid: given two poetic meters p and q we can form a combined meter $p \cdot q$ where a line fits the meter $p \cdot q$ if it can be split into two parts, the first fitting p and the second fitting q. The neutral element will be trivial meter that accepts only empty lines; one could call this "vacumeter". Therefore, if we want to represent poetic meter in Haskell, our representation should probably be a member of the `Monoid` type class.

4 A First Attempt

For our first attempt, we'll just ignore the problem of deviating from the meter, and just worry about whether a line of poetry adheres strictly. We start by defininig a data type to represent individual syllable stresses:

```
data Beat = Stressed | Unstressed
```

Here, we use `Stressed` for — and `Unstressed` for ◡. Since we don't (yet) care about deviations from the meter, we can just represent meters by lists of these values:

```
type Meter = [Beat]
```

For example, iambic pentameter will be this:

```
pentameter =  [Unstressed,Stressed,
  Unstressed,Stressed,Unstressed,Stressed,
  Unstressed,Stressed,Unstressed,Stressed]
```

or alternatively:

```
iamb = [Unstressed,Stressed]
pentameter = iamb `mappend` iamb `mappend`
  iamb `mappend` iamb `mappend` iamb
```

Here, we take advantage of the fact that lists are a member of the type class `Monoid` by default, so `mappend` has already been defined for us.

Given the above, we now want to be able to check to see if a line of poetry matches the required meter. To do this, we'll assume we have some function `vocab :: String -> [[Beat]]` that, given a word, will give us back the list of possible ways that word could be pronounced. (We can't just get back a single way to pronounce it, as some words have multiple pronunciations.) For example:

```
vocab "hello" = [[Unstressed,Stressed]]
vocab "imprecision" = [[Unstressed,Unstressed,
                        Stressed,Unstressed]]
vocab "present" = [[Unstressed,Stressed],
                   [Stressed,Unstressed]]
```

Now, how do we check if a line scans? Well, first we need to compute all possible ways to pronounce that line. We can do this with the following recursive strategy:

- First, use `vocab` to find all ways to pronounce the first word
- Then, find all ways to pronounce the rest of the line
- Finally, combine these two lists of possibilities in all possible ways

The result will be a list of possible pronunciations for the line, each possible pronunciation being a list consisting of one pronunciation for each word in the line. Each word pronunciation will itself be a list of syllable stresses. In other words, we return a list of lists of lists.

We can implement this in Haskell as follows:

```
prons :: [String] -> [[[Beat]]]
prons (word:words) = let ps1  = vocab word
                         pss2 = prons words
                     in [ p1:ps2 | p1  <- ps1,
                                   ps2 <- pss2 ]
prons [] = [[]]
```

Alternatively, for those a little more well-versed in Haskell, we can do this:

```
prons words = mapM vocab words
```

Finally, to check if a line scans, we just check if any possible pronunciation of it matches the meter:

```
scans :: Meter -> [String] -> Bool
scans meter line = any (== meter)
                       (map concat (prons line))
```

5 Making it Fuzzier

So far, so prescriptive. How can we make this looser, allowing for deviations from the meter? Well, the first step is to realise that whether a line scans isn't a rigid, all-or-nothing thing: it's a question of aesthetics, and aesthetics are anything *but* rigid. At the moment, our function `scans` returns a boolean — a yes-or-no value — so clearly, the next step is to make it return something a bit less binary, like a percentage.

```
scans :: Meter -> [String] -> Float
```

Here, we've changed the type of `scans` to return a value of type `Float`, one of Haskell's numeric types. Which exact numeric type doesn't really matter, but one of the advantages of using `Float` is that it can have *fractions*. We can then score our lines on a scale from 0 to 1, with 0 meaning "abominable" and 1 meaning "perfect".

So, in this case, what can we use for our type `Meter`? Well, we can think about what it *is* in terms of what it *does*: given a pronunciation of a line, it needs to tell us how well the line fits the meter. Or, in words a computer can understand:

```
type Meter = [Beat] -> Float
```

Given this, our function `scans` is now reasonably obvious:

```
scans meter = maximum . map meter .
              map concat . prons
```

To put it into words, what we do is find the score for all possible pronunciations and take the maximum. We take the maximum (rather than any other way of combining scores) because we want to assume our poet intended the best possible choice of pronunciations. (It's easier to do it like this than to actually teach a computer to work out which pronunciation was intended.)

Now we can define an iamb in a slightly looser way:

```
iamb [Unstressed,Stressed]   = 1.0
iamb [Stressed,Stressed]     = 0.9
iamb [Stressed,Unstressed]   = 0.8
iamb [Stressed]              = 0.6
iamb [Unstressed,Unstressed] = 0.3
iamb [Unstressed]            = 0.1
iamb _                       = 0.0
```

The specific values here are fairly arbitrary, and could be tweaked depending on what particular task you're trying to perform. The important thing here is that an *actual* iamb gets a perfect score, and its common substitutions get fairly high scores as well, while less common substitutions get lower scores. The final line is a kind of catch-all clause, saying that for any case we didn't cover earlier, we return 0. By providing this kind of definition for all the commonly used metrical feet, we can then allow our poet-programmer to use the monoid operations to build up whatever meter their heart desires.

How do we define the monoid operations? Remember what we said when we first said that meter was a monoid:

- A line fits the meter $p \cdot q$ if it can be split into two parts, the first fitting p and the second fitting q
- The neutral element will be trivial meter that accepts only empty lines

We can use these ideas to make our new `Meter` type a `Monoid`:

```
instance Monoid Meter where
  (p `mappend` q) pron = ...
  mempty pron = ...
```

Let's fill in the blanks. We'll start with `mempty`: it's pretty clear that this will just be a function that only likes empty lines and thinks anything else at all is terrible, like a particularly impatient critic:

```
mempty [] = 1.0
mempty _  = 0.0
```

Next, we define `mappend`. This is a bit more involved: we'll start by working out all the possible ways to split a line into two pieces:

```
splits :: [Beat] -> [([Beat],[Beat])]
splits []     = [([],[])]
splits (b:bs) = ([],b:bs) : [(b:ls,rs) |
                             (ls,rs) <- splits bs]
```

Now we can define `p `mappend` q` for a given line by finding all possible splits of that line into two parts, scoring with `p` on the left part and `q` on the right part, combining the left and right scores and picking the best final score. Like this:

```
(p `mappend` q) line = maximum [ p l * q r |
                                 (l,r) <- splits line ]
```

Here, we combine scores by multiplying them. There are plenty of other ways we could do this, but we want to make sure that two zeros make a zero and two ones make a one. The intuition we want to reflect is this: if the two parts are bad, the whole line is bad, and if the two parts are good, the whole line is good.

6 Scoring by Word

Now we can get a line-by-line score for how well they scan: we've turned our computer into a critic. But maybe this kind of monolithic score isn't that useful if we're just starting to learn about writing poetry, like a teacher that tells you *what* you've got wrong but not *why*. If we've written a bad poem, we'd probably like to know exactly how to fix it. In other words, good criticism is *constructive*.

To accomplish this, we can augment the return type of `scans` with a score for each of the components that make up the line, either words or syllables. We call this data a *fit*:

```
data Fit = F Float [Float]
```

The idea here is to keep track of the contribution made by each syllable to the final score. Once we have these contributions, we can aggregate them into the complete contribution of each word by cross-referencing with the chosen pronunciation for the line. First, we must redefine our monoid for meter to use the new return type:

```
type Meter = [Beat] -> Fit
```

This requires rewriting the definitions of `mempty` and `mappend`. The definition for `mempty` is straightforward: we just augment the original values with a list of zero values equal to the number of syllables given.

```
mempty []  = F 1.0 []
mempty xs  = F 0.0 (replicate (length xs) 0)
```

For `mappend`, we must do two things. First, we must find some operation that combines fits to replace multiplication. This is easy: we just combine the aggregate scores as before, and concatenate the per-syllable scores.

```
combine :: Fit -> Fit -> Fit
combine (F x xs) (F y ys) = F (x*y) (xs ++ ys)
```

Next, we need some way to compare fits, so that we can use the function `maximum`. In Haskell, this is done with the type class `Ord`. Since we already have the aggregate scores, we can simply compare the aggregates:

```
instance Ord Fit where
  compare (F x xs) (F y ys) = compare x y
```

Now, we can redefine `mappend` to deal with fits:

```
(p `mappend` q) line =
          maximum [ combine (p l) (q r) |
                    (l,r) <- splits line ]
```

To implement `scans`, all we need is a function to combine the aggregates by syllables into aggregates by word. This will take a fit and the pronunciation that generated it, and return a new fit. We can do this by iterating over the pronunciation, aggregating as we go.

```
aggregate :: [[Beat]] -> Fit -> Fit
aggregate (p:ps) (F x xs) =
     let r = product (take (length p) xs)
         xs' = drop (length p) xs
         F _ rs = aggregate ps (F x xs')
     in F x (r:rs)
```

Finally, we put a call to this function into the pipeline we defined for `scans` before:

```
scans meter = maximum .
       map (\p -> aggregate p (meter (concat p))) .
          prons
```

Now, if we apply `scans` to a line of poetry, it will not only tell us how well it fits the meter, but *which words* fit particularly well or poorly, showing us how to improve upon our work.

7 Optimising

Our definition of the `mappend` function currently tests all possible ways to split a line in two. In practice, this will result in a great deal of inefficiency, as many of these splits will make no sense: it is unlikely that a four-syllable block could play the role of an iamb, for example. We can avoid some of this inefficiency by packaging each `Meter` with a number representing the longest possible line (measured in number of syllables) that it will fit.

```
type Meter = ([Beat] -> Fit,Int)
```

When combining meters, we can use this length to discard obviously wrong splits:

```
((p,m) `mappend` (q,n)) =
  let f line = maximum [ combine (p l) (q r) |
                         (l,r) <- splitsTo m line ]
```

```
  in (f, m+n)

splitsTo :: Int -> [Beat] -> [([Beat],[Beat])]
splitsTo n []     = [([],[])]
splitsTo 0 xs     = [([],xs)]
splitsTp n (b:bs) = ([],b:bs) : [(b:ls,rs) |
                      (ls,rs) <- splits (n-1) bs]
```

If we assume that `p` will always return a fit of zero for lines longer than `m`, then we should get the same results as we did before, only faster.

8 Evaluation

I have written a web application using the `hpoem` library called the *HPoem Online Bard*, that checks poetry to ensure it is in iambic pentameter. It was compiled with GHCJS and is available online at http://www.cs.nott.ac.uk/~pszjlh/hpoem.html. It runs reasonably well, especially considering it runs entirely in the browser, though loading the vocabulary takes a few seconds and requires a fair amount of memory.

The vocabulary for this program is based on the *CMU Pronouncing Dictionary* [Carnegie Mellon University Speech Group 2015]. Unfortunately, while this dictionary does include stress information, this seems to be based on a different notion of syllable stress to that which poets use. In particular, all single syllable words are considered stressed in this dictionary, even though in practice many can be spoken without stress depending on context. The specification for the meter therefore needed to allow for stressed syllables where unstressed ones would be expected, with the knock-on effect of making the matching a little more liberal than I would have otherwise preferred.

I put a number of sonnets into the program, as well as non-sonnet texts, and it was generally good at highlighting where the lines didn't quite scan. However, with some texts the limitations of the vocabulary became apparent, with archaic words generally being highlighted as questionable. I believe that, given a more appropriate vocabulary file, the results will improve, though for the best results it may be necessary to include some functionality for guessing the pronunciation of unknown words.

9 Conclusion

I have demonstrated how a fuzzy, compositional approach to checking if a given poem scans, based on a monoidal representation of meter. The resulting code is short and easy to understand. I hope that this work will convince readers that this kind of checking need not be particularly sophisticated to produce good results.

I believe there is much more work to be done on fuzzy checking. For example, a Shakespearean sonnet generally includes a "turn" at either the ninth or thirteenth line, where the mood of the poem shifts. Future work could integrate semantic concerns into the checking, to account for forms that have specific requirements on meaning. Furthermore, this work completely ignores rhyme and assonance, focusing only on meter; I believe that integrating this information into the checking ought to be relatively straightforward.

Poetry's monoidal structure comes from the fact that it is essentially linear, but for other forms there may be other structure to exploit. In particular, music is two-dimensional (the dimensions being time and pitch), so perhaps musical form can be treated like a ring or field. Fuzzy matching could also be used for lyrical compositions, to ensure the rhythm of the words fits well with the pitches and rhythmic accents.

Acknowledgements

I would like to thank Johannes Punkt and Rob Mitchelmore for their helpful comments on a draft form of this paper.

References

Geert Adriaens and Dirk Schreors. 1992. From COGRAM to ALCOGRAM: Toward a controlled English grammar checker. In *Proceedings of the 14th conference on Computational linguistics-Volume 2*. Association for Computational Linguistics, 595–601.

Isaac Asimov. 1981. 109. She's No Dope. In *A Grossery of Limericks*. W. W. Norton & Company.

Ranjit Bhatnagar. 2012. Pentametron. (2012). https://twitter.com/pentametron

Botnik Studios. 2016. Voicebox. (2016). http://www.botnik.org

Edwin Brady. 2017. *Type-driven Development with Idris*. Manning.

Carnegie Mellon University Speech Group. 1993–2015. The CMU Pronouncing Dictionary. (1993–2015). http://www.speech.cs.cmu.edu/cgi-bin/cmudict

Gaius Valerius Catullus. 1472. Catullus 16. In *Catullus, Tibullus, Propetius*. Wendelin von Speyer.

Kevin Eastman and Peter Laird. 1984. *Teenage Mutant Ninja Turtles #1*. Mirage Studios.

James E Hoard, Richard Wojcik, and Katherina Holzhauser. 1992. An automated grammar and style checker for writers of Simplified English. In *Computers and Writing*. Springer, 278–296.

Cheng Zhi Anna Huang and Elaine Chew. 2005. Palestrina Pal: a grammar checker for music compositions in the style of Palestrina. In *Proc. of the 5th Conf. on Understanding and Creating Music*. Citeseer.

IBM Programming Research Group. 1954. *Preliminary Report: Specifications for the IBM Mathematical FORmula TRANSlating System, FORTRAN*. IBM.

Justine Kao and Dan Jurafsky. 2012. A computational analysis of style, affect, and imagery in contemporary poetry. In *Proceedings of the NAACL-HLT 2012 Workshop on Computational Linguistics for Literature*. 8–17.

Tom Pankhurst. 2009. ChoraleGUIDE: harmonising Bach chorales. (2009). http://www.choraleguide.com Accessed 2018-06-08.

Marc R Plamondon. 2006. Virtual verse analysis: Analysing patterns in poetry. *Literary and Linguistic Computing* 21, suppl_1 (2006), 127–141.

Donya Quick and Paul Hudak. 2013. Grammar-based Automated Music Composition in Haskell. In *Proceedings of the First ACM SIGPLAN Workshop on Functional Art, Music, Modeling & Design (FARM '13)*. ACM, New York, NY, USA, 59–70. https://doi.org/10.1145/2505341.2505345

William Shakespeare. 1599. *The Most Excellent and Lamentable Tragedie of Romeo and Juliet*. Cuthbert Burby.

William Shakespeare. 1609. Sonnet 116. In *Shakespeare's Sonnets*. Thomas Thorpe.

Dean Keith Simonton. 1986. Aesthetic success in classical music: A computer analysis of 1935 compositions. *Empirical Studies of the Arts* 4, 1 (1986), 1–17.

Dmitrij Szamozvancev and Michael B. Gale. 2017. Well-typed Music Does Not Sound Wrong (Experience Report). In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell (Haskell 2017)*. ACM, New York, NY, USA, 99–104. https://doi.org/10.1145/3122955.3122964

Rui Yan, Han Jiang, Mirella Lapata, Shou-De Lin, Xueqiang Lv, and Xiaoming Li. 2013. i, Poet: Automatic Chinese Poetry Composition through a Generative Summarization Framework under Constrained Optimization. In *IJCAI*. 2197–2203.