Escaping the Prison of Style

Antranig Basman
Raising the Floor, International
London, England
amb26@ponder.org.uk

Philip Tchernavskij
Inclusive Design Research Centre, OCAD University
Toronto, ON, Canada
ptchernavskij@ocadu.ca

ABSTRACT

We seek support for our notion of authorially open programming practices through a critical reading of the notion of "style" in our dominant programming cultures. We argue that this popular notion of programming style is inescapably infused with values which diverge from those of its supposed analogues in areas such as literary and artistic expression. By examining the discourse around and technical distinctions between programming styles, we reveal their fundamentally different nature in structuring dialogues between the writers of computational artefacts and those who experience them. Programming styles are predominantly discussed in inward-facing terms, i.e. with regard only to the experience of programmers with privileged access to the source code. Regardless of their chosen style, programs consist of "imprisoned expressions", built of design elements cut off from access to the program which executes in the world. We construct a miniature integration language, still bounded within the space of existing programming language styles, to solve an open authorship problem, and observe that the increased open ownership of expressions has come at a significant usability cost. We fail to escape the prison, and reflect that our own dialogue remains predominantly inward-facing, due to the lack of proper materials and vocabulary to put communities into correspondence. We look forward to more convivial venues and idioms for expressing computational artefacts, with more equal relationships between the ecologies of construction and ecologies of use.

CCS CONCEPTS

• **Software and its engineering → Reusability**; **Software evolution**; *Collaboration in software development*; *Software notations and tools.*

KEYWORDS

programming styles, programming paradigms, open authorship, software evolution, convivial computing

ACM Reference Format:
Antranig Basman and Philip Tchernavskij. 2020. Escaping the Prison of Style. In *Conference Companion of the 4th International Conference on the Art, Science, and Engineering of Programming ('20), March 23–26, 2020, Porto, Portugal.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3397537.3397548

1 INTRODUCTION

We join this workshop's call for more convivial digital tools, that is, software that "can be easily used, by anybody, as often or as seldom as desired, for the accomplishment of a purpose chosen by the user." [15]. We immediately reflect that conviviality can only be derived from the relations or process surrounding an artefact rather than read off from its directly observable properties. We are concerned by the role existing programming tools and techniques have in excluding large groups of users from convivial relationships mediated by their tools [3, 7]. The vast majority of users of software must choose between off-the-shelf products that may or may not fit their cognitive, physical, and social needs, and are additionally constrained by the fact that they and their collaborators already use many such products that are often incompatible with each other at various levels. Having access to the skills of an expert programmer does little to alleviate these issues, because software is generally constructed in a manner that effectively excludes anyone but its original authors and their peers from participating in the design process. This is less an intentional behaviour on the part of developers, and more an assumption that has become progressively embedded in our tools and best practices through the historical development of the mode of software production [26, section 2.6].

We want to bring about the opposite situation, in which creative networks are empowered to curate, share, modify, and combine digital tools of interest to them at a cost they can afford. Approaches to this goal have been referred to variously as continuing design [11], open authorship [6], and malleable software [26]. In this paper, we apply the lens of open authorship in particular, which is defined by the principle that all programs should remain open to ongoing (re-)design by an evolving network of users. The open authorial principle states that any design decision made by one author can be modified by contributing an additional expression from another author — without the need to rewrite the first author's expression [6]. It implies a programming practice mediated by *external composition*, that is, the ability to superimpose programs by actions taken out in the world, rather than those which require privileged access to source materials. Complex and/or long-lived software typically has several communities of authors and users, whose requirements for the software may diverge over time. External composition has the potential for strengthening conviviality in these circumstances, since these communities should be able to reflect their local requirements in artefacts owned and maintained by themselves. By virtue of moving the point where artefact composition occurs outside the boundary of one development team or organisation, we avoid centring this community as the broker of all relations that are mediated by the artefacts.

Naturally this decentering is accompanied by economics reflecting the relative empowerment or disempowerment of the related communities. Without external composition, variant designs are achieved by means of negotiation around a centralized code base — this is highly costly and excludes marginalized communities of authors and users, whose requirements are irrelevant or counter-productive to the majority. Correspondingly, we believe external composition is an essential mechanism for changing these economics.

This paper is a reflection on how the goals of open authorship fit within the current dialectics of programming practice and computer science, i.e. the distinctions, value judgements, and trade-offs that programmers consider important to their work. The focal point of our reflection is the notion of programming styles.

2 OUR SUBJECT OF REFLECTION

The notion of style in programming has emerged by analogy from other disciplines, such as literature, the visual arts, and arguably more closely related ones such as architecture and civil engineering. The term may be taken in a narrower or a wider sense, more narrowly referring to incidental details of expression such as formatting, naming and granularity of structure, and more widely referring to the primitive mechanisms and organisation of computation.

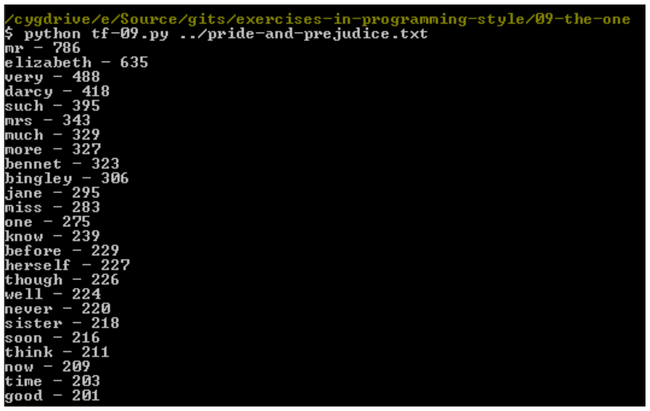
One canonical illustration of the narrow sense is Kernighan and Plauger's *The Elements of Programming Style* [20]. This book – a pastiche of Strunk and White's *The Elements of Style* [16] – is a compendium of maxims primarily aimed at producing programs that are more easily readable by programmers. These maxims are illustrated through a series of narrated refactoring exercises, using code samples taken from programming textbooks. The refactorings are concerned with use of comments, naming, repetition, indirection through sub-procedures, etc. Kernighan and Plauger thus frame style as communicative clarity, something that a given program can have more or less of.

In her book *Exercises in Programming Style* [22], Lopes surveys programming styles in a wider sense. She goes beyond the kind of bookkeeping details mentioned above and looks at more substantial details of how computations are organised and expressed. In her interpretation of "style", Lopes combines notions such as models of computation, language paradigms, and architectural patterns. For example, there are styles representing programming with stack machines, spreadsheets, and Haskell-esque monads. Lopes argues that these phenomena in practice overlap and interact:

> *[T]here is a continuum in the spectrum of how to write programs that goes from the concepts that the programming languages encourage/enforce to the combination of program elements that end up making up the program; languages and patterns feed on each other, and separating them as two different things creates a false dichotomy. [22, p. xii]*

Lopes' survey consists of set of sample Python programs written in different styles to solve the same task[1] and accompanying descriptions of each style, discussions of their trade-offs, and suggested programming exercises, such as comparing different styles, or extending one of the samples with additional behaviour [22]. The chosen task is to load a text as input and produce as output a list of the 25 most frequent words in the text, in descending order of frequency.

[1]At the time of writing, 41 different styles are collected at https://github.com/crista/exercises-in-programming-style



```
/cygdrive/e/Source/gits/exercises-in-programming-style/09-the-one
$ python tf-09.py ../pride-and-prejudice.txt
mr - 786
elizabeth - 635
very - 488
darcy - 418
such - 395
mrs - 343
much - 329
more - 327
bennet - 323
bingley - 306
jane - 295
miss - 283
one - 275
know - 239
before - 229
herself - 227
though - 226
well - 224
never - 220
sister - 218
soon - 216
think - 211
now - 209
time - 203
good - 201
```

Figure 1: An example of one of the sample programs from [22] in use, here style 9, "The One"[2]

We consider Lopes' book and survey to be an excellent illustration of what the phenomenon of programming styles encompasses for the majority of computer scientists and programmers today. We believe it does a fair job of reflecting the trade-offs and value judgements that actual programmers consider in justifying one style over another. Thus, it forms a good basis for our exercise, to discuss what these reveal about programming as an authorial practice.

3 A SEEMING DIVERSITY, A COMMON PROBLEM

Figure 1 illustrates the view of one of Lopes' sample programs "from the outside". It receives a text file from standard input and prints the computed word frequencies. All the programs in [22], from the point of the user, are exactly the same one, not merely variants. They are invoked in just the same way and produce identical output. The programs not only have identical interfaces for execution, but identically non-existent interfaces for modification. In each case, the programs are designed in a way that precludes anyone encountering the running program from attempting to understand, change, or integrate it.

We call this a paradigm of "imprisoned expressions", because the design elements of individual program are cut off from extension or reuse by anyone not already inside the boundary of the file. The diversity of styles on offer indicates an apparent freedom to choose one thing or another, yet our choices are framed in such a way that this space of possibilities in practice feels like a prison. We can choose how we want to express our computational power, but we cannot choose how that power is shared with the wider world.

This situation follows from the framing of the style exercise, but it also reveals the authorial culture of programming. These are the styles of an inward-facing discipline, i.e. a practice of programming that is primarily concerned with the work of producing

[2]https://github.com/crista/exercises-in-programming-style/blob/master/10-the-one. Note that we refer to the style numbers and names used in the book [22], which do not always match the ones used in the GitHub repository.

initial, correct, and efficient programs, excluding the experiences and labour of subsequent (re-)users, and implicitly assuming that role of enacting reuse must take place within the discipline.

The effect of imprisoned expressions is that the ecology of function — the network of people and tools surrounding the running programs — is cut off from any direct relation with the ecology of design – the corresponding network involved in the creation of the source code [8]. These are "styles" only evident to the elite construction ecology.

4 THIS IS A DILUTED NOTION OF STYLE

Compare this with Christopher Alexander's presentation of style in his architectural design patterns [9], familiar to programmers through their misappropriation by Gamma et al. [12]. For example, his "Six Foot Balcony" pattern:

> *167 Six Foot Balcony: A balcony is first used properly when there is enough room for two or three people to sit in a small group with room to stretch their legs, and room for a small table where they can set down glasses, cups, and the newspaper. No balcony works if it is so narrow that people have to sit in a row facing outward. The critical size is hard to determine, but it is at least six feet. [9]*

Note the inescapable invocation of the ecology of use. This pattern describes the design choice in terms of its usefulness and habitability to the eventual users of the building.

Furthermore, this pattern coexists alongside:

> *163 Outdoor Room: . . . a partly enclosed space, outdoors, but enough like a room so that people behave there as they do in rooms, but with the added beauties of the sun, wind, and smells, and rustling leaves, and crickets. [9]*

Note that these are overlapping, opportunistically characterised elements — they may coexist partially, overlapping, or not at all. Contrast this to programming "styles" and "patterns", which are purely inward-looking to the ecology of production, and also largely taxonomical — invocation of one pattern applied to a particular design element typically rules out other patterns applicable to that element (although naturally patterns can be invoked in cooperating relations on cooperating design elements). By contrast, programming styles — and programming design patterns — are used to end conversations through classification, not begin them through inspiration. They are variant "approaches to a problem" rather than members of a vocabulary.

Our critique of the invocation of "style" in programming turns out to echo Alexander's own feelings about the Design Patterns community. In a keynote delivered to OOPSLA in 1996, Alexander [1] decried their moral deframing of pattern languages and their lack of orientation toward the communities that motivate and live with programs, and described programmers as behaving as "guns for hire":

> *My comment on this? Please forgive me, I'm going to be very direct and blunt for a horrible second. It could be thought that the technical way in which you currently look at programming is almost as if you were willing to be "guns for hire." In other words, you are the technicians. You know how to make the programs work. "Tell us what to do daddy, and we'll do it." That is the worm in the apple. [1]*

More than 20 years later, nothing has changed, and no shred of moral orientation has been introduced into our notion of "style". Let us make the 2020s the decade when we finally try to deliver on some of Alexander's exhortation to produce living, open, convivial structures, rather than inward-looking, technocratic and dead ones:

> *What I am proposing here is something a little bit different from that. It is a view of programming as the natural genetic infrastructure of a living world which you/we are capable of creating, managing, making available, and which could then have the result that a living structure in our towns, houses, work places, cities, becomes an attainable thing. That would be remarkable. It would turn the world around, and make living structure the norm once again, throughout society, and make the world worth living in again.*
> *This is an extraordinary vision of the future, in which computers play a fundamental role in making the world — and above all the built structure of the world — alive, humane, ecologically profound, and with a deep living structure. [1]*

5 THERE ARE WORSE THINGS THAN A CATALOGUE OF STYLES

Despite our reservations on the notion of style it embodies, we welcome the catalogue of program styles assembled in Lopes [22] since it genuinely shows our field at its best. Travel, even travel within a prison, broadens the mind[3], and the catalogue is a helpful assistance to the inhabitants of the prison to desist from their traditional activities of mounting a power struggle with their neighbours imprisoned in the adjoining cell. Many programmers can live out their careers either unaware that substantially differing styles exist, or else locked in an endless struggle to demonstrate that the power granted by its furnishings to enslave visitors to their cell greatly exceeds that of their colleagues. Notable examples include Hoyte's declaration "macro programming is, of course, not about style. It is about power" [14], and Graham's sneering at the users of a programming language he invidiously dubs "Blub" and their incapability of grasping the superior power of his chosen style [13]. These inhabitants are what Orwell [24] would call "power-worshippers" — they worship the strong simply because they are strong, rather than for their tendency to lead their users to "breathe the air of equality" [23].

So let us by all means celebrate the map of our prison provided in the catalogue of styles, and see if, with its help, we can plot out what it might take to stage an escape.

[3]"Denmark's a prison." — "Then is the world one", Hamlet: Act II, Scene 2

6 IN PART OF THE PRISON, THERE IS MORE LIGHT

Though the styles surveyed by Lopes are primarily motivated in inward-facing terms, scattered throughout there are styles with values oriented toward enabling wider authorial participation. In some cases, these values are obscured by the fact that the survey opts to express all styles through Python source code invoked from the command line. For example, style 1, "Good Old Times"[4] reflects the constraints of programming early computers with minimal primary memory, and style 26, "Spreadsheet"[5] represents different stages of data processing as columns connected by formulas. These styles enjoyed useful virtues of *externalisation* in their original use contexts. Externalisation is the ability to expose a running program's behaviour and state in a document form that can be freely exchanged and modified. It is a key value for living, convivial software, because it supports ongoing (re-)design and integration [10].

In the former case, this arose through the natural virtues inherited from the embodiment of computation in the physical world of wires and memory locations, which had not yet been effaced by decades of programming language refinement. In the latter, the virtue had been explicitly designed into the interaction structure — a coordinatised, reactive surface of data was exposed to the user as the primary interaction idiom, a direct window onto the system's internal state facing all the way out. In both cases, these virtues are lost in the traditional dogmatic (dis)association between a programming language's variables and the resulting completely opaque system state.

In other cases, styles directly address the historical need to support unpredictably evolving software. Style 14, "Hollywood"[6], and Style 15, "Bulletin Board"[7]represent two different approaches to inversion of control-style architectures. Lopes recognises in their writeup that they are suited to the design of systems whose evolution cannot fully be foreseen:

> *Publish-subscribe architectures are popular in companies with large computational infrastructures, because they are very extensible and support unforeseen system evolution – components can be easily added and removed, new types of events can be distributed, etc. [22, p. 119]*

However, fuller recognition that these styles might enable the possibility for open authorship, which must not require community members to rewrite the expressions of other authors in order to use them in designs over which they have full ownership [6], is relegated to the reader exercises, for example exercise 14.2:

> *Exercise 14.2: Words with z. Change the given example program so that it implements an additional task: after printing out the list of 25 top words, it should print out the number of non-stop words with the letter z. Additional constraints: (i) no changes should be made to the existing classes; adding new classes and more lines of code to the main function is allowed; (ii) files should be read only once for both term-frequency and "words with z" tasks. [22, p. 114]*

Style 15 is presented as a "logical end point" of style 14, but it is far from an endpoint, since the design still retains a single point of design orchestration expressed within unmodifiable program code, where the components solving sub-tasks are wired together:

```
em = EventManager()
DataStorage(em), StopWordFilter(em), WordFrequencyCounter(em
    ↪ )
WordFrequencyApplication(em)
em.publish(('run', sys.argv[1]))
```

Note that the corresponding exercise to 14.2 in this section, 15.2, still permits the traditional evasion "adding more lines of code to the main function is allowed".

7 STYLE 15B - "WEAK TEA"

Let's imagine a variant style, we'll label 15b, continuing with this design intention towards open authorship. This style should allow us to solve both the initial word counting problem and the variant "words with z" problem, while satisfying the open authorial constraint of creating the variant program only by means of external composition.

We place this style at position 15b in the index, immediately after the "Bulletin Board" style 15 that represents the high point of authorial openness achieved in the taxonomy. Our program still lies within the convex hull of our prison, combining elements seen in several different styles surveyed by Lopes, such as the monadic composition styles 9, "The One" and 24, "Quarantine"[8] and the reflective facilities broadly surveyed in chapters part V of the book (although not the same ones in detail[9]), as well as the configuration language style seen in style 19, "No Commitment"[10].

However, it orchestrates these styles to a particular end — to push the goals of open authorship closer to the extreme possible within the standard Python ecosystem. In particular, we want to deliver on the endpoint of the goals we see underlying exercises 14.2 and 15.2, which as an open authorial narrative as per [6] we could describe as "Author *A* has written a term frequency application. Author *B* wishes to use all of *A*'s design to meet a closely related goal, but *A* is not entitled by a community relationship to modify *A*'s code, and also does not want to do work proportional to the size of *A*'s design or their enclosing community".

Our goals lead us to reconstruct some elements from [6] in the context of this small ecology of Python programs. Firstly, we construct a minimal configuration language (or *integration language*, in the sense of [18]) expressed as a hash of JSON records determining the dataflow and sequencing of the Python functions in *A*'s design, similar in construction and intention to parts of Infusion [5].

[4]https://github.com/crista/exercises-in-programming-style/blob/master/01-good-old-times
[5]https://github.com/crista/exercises-in-programming-style/blob/master/27-spreadsheet
[6]https://github.com/crista/exercises-in-programming-style/blob/master/15-hollywood
[7]https://github.com/crista/exercises-in-programming-style/blob/master/16-bulletin-board
[8]https://github.com/crista/exercises-in-programming-style/blob/master/25-quarantine
[9]These reflective faculties are seen in three example styles, 16 "Introspective" 17 "Reflective" and 18 "Asides" but in the first, they are merely used to create obstructions to execution, in the second they are used to further obscure the program's source text from the ecology of construction, and in the third they implement an extraneous piece of functionality (profiling) only of interest to technicians.
[10]https://github.com/crista/exercises-in-programming-style/blob/master/20-plugins

An integration language is a specialisation of the notion of a configuration language presented by Lopes, i.e. a declarative language for specifying choices among a set of options, e.g. module implementations and constructor parameters[11]. Integration languages aim to specify all inter-module interfaces, increasing the possibility of reusing modules created by different authors without modification [18]. For example, whereas the configuration language used in “No Commitment” (INI files) [22, p. 144] maps module names specified in a template program to particular implementations in the base language of Python, our example integration language is used to specify the choice of base language functions, as well as their sequencing and dataflow. This obviates the need for a base language template program. The dataflow specification is achieved through a minimal selector dialect expressed with respect to the naming structure of the integration language (its hash keys) — such a selector dialect is also a design element predicted by [6].

Secondly, we define a “program addition operator” $\oplus$ [6, section 5.3] that is capable of fusing multiple programs expressed in our integration language together, allowing for the expression of program additions as separate programs. Due to the alignment properties of our integration language, this operator can be expressed by a simple dictionary merge. The structure of the configuration language is designed entirely to allow the program addition operator to serve the function of external composition. Whilst, computationally, we implement the behaviour of the operator as a standard function in the Python program, it could be mechanically operated from outside the system by merging the structure of the JSON documents representing A and B’s design, without recourse to any of the facilities of the programming language or its runtime.

Our resulting “minimal viable integration language” occupies about 50 lines of Python and is not reproduced here[12], but the program itself is shown in Listing 1 and the corresponding configuration structure is shown in Listing 2.

Note that listing 1 is extremely similar to the implementation in [22] functional style 9, “The One” — the only substantive difference is that we have taken the opportunity to shift out meaningfully parameterisable constants such as i) the stop-word file, ii) the sort order, iii) the number of high frequency terms to display into function arguments and hence out into the configuration dialect.

7.1 About the Integration Language

Our integration language implements a small dialect of selectors used to express the structural connections conventionally expressed by e.g. function executions and shared variables. It makes use of three forms of selectors (listing 2), the first in `priority` fields to control execution order using positional constraints such as `after:frequencies`, and the second in interpolated arguments in `args` such as `$sort` to control dataflow, and the third to make fully

[11]Lopes briefly discusses the historical development of configuration languages into Architecture Description Languages, which are more similar to our notion of integration languages [22, p. 144–146].

[12]Full source code for this sample is available at https://github.com/amb26/exercises-in-programming-style/tree/weak-tea/15b-weak-tea, in particular with the integration language implemented at https://github.com/amb26/exercises-in-programming-style/blob/weak-tea/15b-weak-tea/tf_config.py

```python
#!/usr/bin/env python
import sys, re, operator, string, tf_config

# The functions

def read_file(path_to_file):
    with open(path_to_file) as f:
        data = f.read()
    return data

def filter_chars(str_data):
    pattern = re.compile('[\W_]+')
    return pattern.sub('␣', str_data)

def normalize(str_data):
    return str_data.lower()

def tokenize(str_data):
    return str_data.split()

def remove_stop_words(word_list, stop_words_file):
    with open(stop_words_file) as f:
        stop_words = f.read().split(',')
    # add single-letter words
    stop_words.extend(list(string.ascii_lowercase))
    return [w for w in word_list if not w in stop_words]

def frequencies(word_list):
    word_freqs = {}
    for w in word_list:
        if w in word_freqs:
            word_freqs[w] += 1
        else:
            word_freqs[w] = 1
    return word_freqs

def sort(word_freq, reverse):
    return sorted(word_freq.items(), key=operator.itemgetter
        ↪ (1), reverse=reverse)

def top_freqs(word_freqs, count):
    top25 = ""
    for tf in word_freqs[0:count]:
        top25 += str(tf[0]) + '␣-␣' + str(tf[1]) + '\n'
    return top25

# The main function

config = tf_config.LoadConfig('tf_15b.json')
# Prevent the config from actually executing if this program
      ↪ is not the top-level script
if (__name__ == '__main__'):
    tf_config.ExecuteConfig(config, sys.argv[1])
```

Listing 1: `tf_15b.py`: Base language functions of style 15b expression of term frequency program

```
{
    "steps": {
        "read_file": {
            "func": "tf_15b.read_file",
            "args": ["$directArg"],
            "priority": "first"
        },
        "filter_chars": {
            "func": "tf_15b.filter_chars",
            "args": ["$read_file"],
            "priority": "after:read_file"
        },
        "normalize": {
            "func": "tf_15b.normalize",
            "args": ["$filter_chars"],
            "priority": "after:filter_chars"
        },
        "tokenize": {
            "func": "tf_15b.tokenize",
            "args": ["$normalize"],
            "priority": "after:normalize"
        },
        "remove_stop_words": {
            "func": "tf_15b.remove_stop_words",
            "args": ["$tokenize", "../stop_words.txt"],
            "priority": "after:tokenize"
        },
        "frequencies": {
            "func": "tf_15b.frequencies",
            "args": ["$remove_stop_words"],
            "priority": "after:remove_stop_words"
        },
        "sort": {
            "func": "tf_15b.sort",
            "args": ["$frequencies", true],
            "priority": "after:frequencies"
         },
         "top_freqs": {
             "func": "tf_15b.top_freqs",
             "args": ["$sort", 25],
             "priority": "after:sort"
         },
         "print_freqs": {
             "func": "print",
             "args": ["$top_freqs"],
             "priority": "after:top_freqs"
         }
    }
}
```

Listing 2: `tf_15b.json`: Configuration language expression accompanying base language functions in Listing 1

externalisable, module-qualified references in `func` to the implementation functions using expressions such as `tf_15b.frequencies`. This leads it to be extremely verbose.

Note that in the extremely compact functional styles such as style 9, "The One" and style 24, "Quarantine", as well as the basic style 6, "Candy Factory"[13], we get all of this topology for free as part of the natural binding structure induced by function composition. This candy is indeed sweet until we find ourselves faced with a differential design exercise which requires us to fish an intermediate computation out of the chain for the benefit of an unsuspected author.

The "words with z" task is only a rudimentary external composition task. Many apparently ordinary occurrences of program extension require more complex authorial manoeuvres than intercepting intermediate computations in a chain of functions, for example introducing or removing layers of containment in a data structure. Such cases present more complex issues for defining suitable integration language mechanisms. Some of these issues and their historical solutions are reviewed more systematically in [6].

```
#!/usr/bin/env python
import sys, tf_config, tf_15b

def filter_words(words, substring):
    return [word for word in words if substring in word]

config = tf_config.LoadConfig('tf_zwords.json')
if (__name__ == '__main__'):
    tf_config.ExecuteConfig(config, sys.argv[1])
```

Listing 3: `tf_zwords.py`: Base language functions of author B's addition to A's design in Listing 1

7.2 The Differential Design

Let us now present the differential part of the design — author *B*'s addition to the configuration and program written by *A*. As promised, *B* includes *A*'s program unmodified, and hence their configuration, using the existing Python `import` facility, and grafts their sequence points onto the end of *A*'s sequence whilst at the same time fishing out an intermediate computation of *A* as returned by the `frequencies` step. *B*'s only requires to implement one function, `filter_words`, capable of filtering a list of strings for those including a given string, in their driver, shown in Listing 3.

Finally we show author *B*'s configuration language expression in Listing 4, which accompanies the base language expression in Listing 3.

[13]https://github.com/crista/exercises-in-programming-style/blob/master/06-pipeline

```
{
    "parent": "tf_15b.config",
    "steps": {
        "filter_words": {
            "func": "tf_zwords.filter_words",
            "args": ["$frequencies", "z"],
            "priority": "after:print_freqs"
        },
        "print_words": {
            "func": "print",
            "args": ["$filter_words"],
            "priority": "after:filter_words"
        }
    }
}
```

Listing 4: `tf_zwords.json`: Configuration expressions of author B's addition to A's design in Listing 3

Interesting features in Listing 4 are:

- The `parent` definition referencing author A's config as `tf_15b.config`. This indicates declaratively that B's program is to be composited on top of A's. This could have been inserted into the driver definition in the base language code, but we are indicating the path towards authoring managed by external tools by trying to retain a standard boilerplate in each file.
- The configuration meets the differential authorship challenge of sequencing execution after A's program with the `after:print_freqs` priority, whilst reaching into the middle of A's dataflow with the `$frequencies` reference.

Another interesting feature can be observed together with the snippet of the configuration language implementation shown in Listing 5. This is the site where two configurations are combined, and it has a very simple form — the dictionaries of "steps" are simply superimposed. It's a crucial goal of our dialect that this composition process takes the simplest possible form, a mechanical process that always succeeds in producing a valid program that expresses the appropriately combined intent of the two authors. Without meeting this goal, it is infeasible that the composition process could be enacted outside the ecosystem of the tool chain and runtime. However, our achievement of this goal is purely moral, since we have not succeeded in producing any such venue outside the system where the composition could plausibly take place. Also, the output of our addition system is still imprisoned within the boundaries of the Python ecosystem — we cannot meaningfully combine programs before the relevant module has been referenced via an `import` statement. Section 8.2 speculates on routes out of the prison that necessarily stray into the operating system's notion of the structure of dynamically loadable objects, and [4] describes the "Nexus" system built on top of Infusion that permits program addition to occur via HTTP verbs.

```
    # This line implements the "program addition operator"
    config['steps'].update(parentConfig['steps'].copy())
```

Listing 5: Snippet of `tf_config.py`: Site of the program addition operator ⊕

8 SEVERAL CONCRETE LOSSES, AND A COUPLE OF INTANGIBLE GAINS

On quite a number of important fronts, our stylistic re-expression of the term frequency task in section 7 represents a clear loss. As well as requiring the implementation of a "mini-framework" which slightly exceeds in complexity the actual target programming task, the resulting expressions are extremely verbose (at least twice the length of the standard functional samples, and orders of magnitude longer than the coding golf samples), feature poor locality of design reference, where the reader has to study several separated design elements simultaneously in order to understand the design, and feature numerous fragile linkages where design elements are connected together by fairly long strings which are easy to mistype and misread.

What is worse, we have put a substantial part of the design outside the reach of the vital tool chain which supports the standard programmer. Other than being linted as being a valid JSON file, the configuration language expressions can't be easily checked for validity and consistency, and errors which occur in them end up being highlighted in unilluminating parts of the framework code rather than in the design element causing the error.

Why on earth have we done this?

At this stage, the design victories of the "Weak Tea" style are largely moral. We've provided an arena in which the expressions of different authors can be combined, without privileging any of them as central, or relying on the machinery of entire programming languages and their toolchains. Our integration language is significantly constrained compared to languages capable of arbitrary Turing-complete computation, improving the likelihood of creating structured editing tools and substantially different visual or non-visual representations of it. The infelicities of expression, the duplication of selector names and poor locality of reference could be "folded up" by suitable tools which, for example, can show the long expanded chain in Listing 2 as equivalent to its compact functional binding equivalent when the view of wider authorial affordances is not necessary.

However, these tools, whilst in theory easier to build on top of this representation, do not exist, and would require a really substantial implementation effort. There is little history of communities effectively building such things — even at its height, the Spring Framework, one of the best-attested and supported configuration languages, only enjoyed rudimentary authoring support. And a truly effective integration language will need a far richer dialect than the ones we've been able to describe so far.

8.1 Imagining a Community

Equally damning is imagining a community for which representations such as listing 1 could be useful. It's very hard to imagine an author capable of successfully editing such a prolix and fragile JSON representation without the help of powerful tools, who wouldn't be more capable of editing straightforward binding chains in Python itself such as

```
print_all(sort(frequencies(remove_stop_words(scan(
    ↪ filter_chars_and\_normalize(read_file(sys.argv[1]))
    ↪ )))[0:25])
```

as seen in style 5, "Candy Factory". In any case, the expressions to be duplicated are so short compared to the openly authorable representation that economics strongly favour simply duplicating them. Designs need to be very large, communities very extensive, and the tools brought to play very powerful, before we can imagine the economics shifting credibly onto the other side. But we need to imagine this, if we mean to establish a broad literacy supporting convivial relations between the expressions of unbounded communities.

8.2 Expanding the Coverage

One can imagine communities for which the coverage offered by our example configuration language might be useful, but in practice this has not carried us terribly far. To start with, as we allude in section 7.1, the authorial flexibility we've achieved is quite limited — programs consuming streams of values connected via pipes are capable of a wide range of tasks, as the corresponding ecology of UNIX pipes shows, but in practice real communities very often require much more power to organise the allocation and deallocation of state, the naming and correspondence of different pieces of state in unrelated or nested collections, reference to historyful values, etc. We need a much more ambitious system for accounting for the contents of memory and establishing such correspondences, as sketched in [18, 19].

There is an important subtlety we've so far glossed over in our notion of "composition", which is the geometry with which it occurs. An important restriction with previous notions of composition (e.g. those of Backus [2]) is not just the one we have stressed so far, that the composition occurs within the privileged realm of those in the community who change source code, but that the composition which it enacts only joins programs together at their *edges* — that is, classically, where functions accept arguments, or structures contain members. We could call this *peripheral composition*. The ecology of UNIX pipes is a paradigmatic exemplar of this style of composition — whereas as we've shown, even in this rudimentary task, we need to do more, and instead align entire programs across their *surfaces*, blending the complete intention of two authors. This appeals to a notion of design which may be named *phenotropic*, following Lanier [21], where we consider systems to be open for interaction across their entire extent. This is in contrast to other dominant design metaphors such as the cellular model of Kay [17] and the whole message-passing idiom of which it is a part, where design elements are expected to be insulated by mostly impermeable membranes.

An important element, therefore, of external composition promoting convivial interactions is that it comprise not only ordinary peripheral composition, and phenotropic composition, but also perhaps other richer means of combining the intentions of multiple authors into a single design.

8.3 Actually Externalising the Design

Moreover, we have failed to deal with many of the practical issues and articulation work raised in the surrounding ecosystem. We have failed to account for how the theoretically highly legible JSON configuration files are practically shipped around along with the computational artefacts they describe, and how the naming system they operate is intended to interact with the existing ones supplied by the Python ecosystem, let alone those of unrelated programming languages. For example, a substantial source of fragility is our reliance on the Python runtime's table of loaded modules, indexed by the names they are given as imports. In practice, this relies on all kinds of unstable details such as the directory structure of the environment holding the source code, and the exact versions of code found there. It's difficult to imagine remedies for this that don't make our already prolix expressions bristle with ever-longer qualified names for the things they are referencing.

In practice, the standoff between conventional programming languages and integration languages will always be unstable, and we don't see a credible escape route that doesn't leave conventional programming languages mostly demolished. In their role as base language for a truly effective integration domain, it seems that only a small subset of their current capabilities can be supported. Further research and implementation may end up changing this view.

Style 19, "No Commitment" shines light in a helpful direction for extension. The plugin model, depending on facilities supplied by the base operating system, is a practical model for external reuse. Unfortunately, the operating system supplies extremely rudimentary facilities for decoding the ontology of loaded objects — in practice, named external function entry points into modules, and if you are lucky, a calling convention suitable for invoking them. In terms of supplying a full ontology for the contents of memory, including the layout of memory structures, their site and reason for allocation, the official OS metadata is inadequate, although [19] has noted that for practical purposes, many operating systems operate a far richer unofficial bus of metadata in order to support advanced authorial affordances such as debugging and profiling. Our vision of open authorship implies that every dynamically loadable object in the OS could be inspected for its complement of configuration language making all such of its capabilities fully visible.

8.4 What Lies Immediately Outside the Prison

In terms of a practical extension of the immediate exterior of our prison, we could imagine extending the configuration dialect of style 19, "No Commitment" into a richer system of metadata by which modules advertise sites capable of allocation in structured forms such as those seen in style 12, "Closed Maps"[14], and express relations in a symbolic form of selectors so that the boilerplate seen in listings such as `tf-18.py` in section 18.2 can be eliminated. However, Lopes provides a salutary warning that correctly predicts that simplistic and incomplete attempts to solve the problem of open authorship will initially make the problem they are trying to solve much worse rather than better:

[14]https://github.com/crista/exercises-in-programming-style/tree/master/13-closed-maps

> *Modern frameworks have embraced this style of programming for supporting usage-sensitive customizations. However, when abused, software written in this style can become a "configuration hell", with dozens of customization points, each with many different alternatives that can be hard to understand. Furthermore, when alternatives for different customization points have dependencies among themselves, software may fail mysteriously, because the simple configuration languages in use today don't provide good support for expression of dependencies between external modules.* [22, p. 145]

In practice, expression of correlated dependencies is just one of many ambitious problems such a system needs robust solutions to. As well the previously mentioned problems of supplying an ontology for the contents of memory at any moment, and clear isolation of sites capable of issuing I/O, the system most importantly needs to track the provenance of every expression entered into it, so that any observed effect can be reliably traced back to the expressions which gave rise to it.

9 STRUCTURE OF AN ESCAPE

A metaphor for the structure of the escape from such a prison was given by Idries Shah in a Sufi teaching story appearing in his "The Magic Monastery" [25], which we reproduce *in extenso*:

> *A man was once sent to prison for life, for something which he had not done.*
> *When he had behaved in an exemplary way for some months, his jailers began to regard him as a model prisoner.*
> *He was allowed to make his cell a little more comfortable; and his wife sent him a prayer-carpet which she had herself woven.*
> *When several more months had passed, this man said to his guards:*
> *"I am a metalworker, and you are badly paid. If you can get me a few tools and some pieces of tin, I will make small decorative objects, which you can take to the market and sell. We could split the proceeds, to the advantage of both parties."*
> *The guards agreed, and presently the smith was producing finely wrought objects whose sale added to everyone's well-being.*
> *Then, one day, when the jailers went to the cell, the man had gone. They concluded that he must have been a magician.*
> *After many years when the error of the sentence had been discovered and the man was pardoned and out of hiding, the king of that country called him and asked him how he had escaped.*
> *The tinsmith said:*
> *"Real escape is possible only with the correct concurrence of factors. My wife found the locksmith who had made the lock on the door of my cell, and other locks throughout the prison. She embroidered the interior designs of the locks in the rug which she sent me, on the spot where the head is prostrated in prayer. She relied upon me to register this design and to realize that it was the wards of the locks. It was necessary for me to get materials with which to make the keys, and to be able to hammer and work metal in my cell. I had to enlist the greed and need of the guards, so that there would be no suspicion. That is the story of my escape."*

10 CONCLUSION

We have argued that computational expressions are imprisoned as a result of the structure of languages, tools and idioms underlying today's software construction, and are unable to participate in an open ecology of function where communities have the power to effectively and economically own their software. We have recognised that a catalogue of today's programming styles is a useful starting point to map out the structure of that prison and to seek out weak points from which an escape might be launched. We have constructed a miniature integration language best meeting the needs of open authorship from within the prison of the Python language and found that it offers a substantially poorer authorial experience in most concrete aspects than the use of conventional styles. We have planned to construct increasingly ambitious such languages, and the crucially necessary accompanying authoring tool supports, until we overcome the extremely substantial obstacles preventing an escape into the apparently extremely hostile exterior of the prison. And perhaps, after centuries, or millenia, the correspondence we succeed in establishing between creators and users of software might give rise to the possibility for genuine, communicative, styles to emerge.

ACKNOWLEDGMENTS

The authors are indebted to Colin Clark for illuminating conversations leading to some of the ideas in this paper, and for the thoughtful critique he delivered at its presentation. This critique may be read at https://wiki.fluidproject.org/display/fluid/Presentations under the title of "The Question Concerning Style".

REFERENCES

[1] Christopher Alexander. 1996. Keynote Speech to the 1996 OOPSLA Convention. http://www.patternlanguage.com/archive/ieee.html
[2] John Backus. 1978. Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. 21, 8 (1978), 613–641.
[3] Antranig Basman. 2017. If What We Made Were Real. In *Proceedings of the 28th Annual Workshop of the Psychology of Programming Interest Group (PPIG 2017)*.
[4] Antranig Basman, Luke Church, Clemens Klokmose, and Colin Clark. 2016. Software and How it Lives On - Embedding Live Programs in the World Around Them. In *Proceedings of the 27th Annual Workshop of the Psychology of Programming Interest Group (PPIG 2016)*.
[5] Antranig Basman, Colin Clark, and Clayton Lewis. 2015. Harmonious Authorship from Different Representations. In *Proceedings of the 26th Annual Workshop of the Psychology of Programming Interest Group (PPIG 2015)* (Bournemouth, England).
[6] Antranig Basman, Clayton Lewis, and Colin Clark. 2018. The Open Authorial Principle: Supporting Networks of Authors in Creating Externalisable Designs. In *Proceedings of the 2018 ACM OOPSLA* (Boston, MA, USA) *(Onward! 2018)*. ACM, New York, NY, USA, 29–43.
[7] Antranig Basman and Philip Tchernavskij. 2018. What Lies in the Path of the Revolution. In *Proceedings of the 29th Annual Workshop of the Psychology of Programming Interest Group (PPIG 2018)*.
[8] Antranig Basman, Philip Tchernavskij, Simon Bates, and Michel Beaudouin-Lafon. 2018. An Anatomy of Interaction: Co-occurrences and Entanglements. In *Companion to the Second <Programming>* (Nice, France) *(Programming '18)*. ACM, New York, NY, USA, to appear.
[9] Christopher Alexander. 1977. *A Pattern Language.* Oxford University Press.
[10] Colin Clark and Antranig Basman. [n.d.]. Tracing a Paradigm for Externalization: Avatars and the GPII Nexus.
[11] Colin Clark and Sepideh Shahi. 2018. On Continuing Creativity. In *Proceedings of the Psychology of Programming Interest Group (PPIG 2018)*.
[12] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley.
[13] Paul Graham. 2003. Beating the Averages. http://www.paulgraham.com/avg.html
[14] Doug Hoyte. 2008. *Let Over Lambda.* Lulu.com.
[15] Ivan Illich and Anne Lang. 1973. *Tools for conviviality.* Harper & Row New York.
[16] William Strunk Jr. and Elwyn Brooks White. 1959. *The Elements of Style.* Macmillan, New York.

[17] Alan Kay. 2003. Clarification of "object-oriented". https://userpage.fu-berlin.de/~ram/pub/pub_jf47ht81Ht/doc_kay_oop_de

[18] Stephen Kell. 2011. The Mythical Matched Modules: Overcoming the Tyranny of Inflexible Software Construction. In *Proceedings of the 24th ACM OOPSLA* (Orlando, Florida, USA) *(OOPSLA ’09)*. ACM, New York, NY, USA, 881–888.

[19] Stephen Kell. 2015. Towards a Dynamic Object Model within Unix Processes. In *Proceedings of the 2015 ACM Onward!* (Pittsburgh, PA, USA) *(Onward! 2015)*. ACM, New York, NY, USA, 224–239.

[20] Brian W. Kernighan and P. J. Plauger. 1982. *The Elements of Programming Style.* McGraw-Hill, Inc.

[21] Jaron Lanier. 2003. Why Gordian software has convinced me to believe in the reality of cats and apples. https://www.edge.org/conversation/jaron_lanier-why-gordian-software-has-convinced-me-to-believe-in-the-reality-of-cats

[22] Cristina Videira Lopes. 2014. *Exercises in Programming Style.* Chapman & Hall/CRC.

[23] George Orwell. 1938. *Homage to Catalonia.* Secker and Warburg.

[24] George Orwell. 1946. Second Thoughts on James Burnham. *Polemic* (1946).

[25] Idries Shah. 1972. *The Magic Monastery.* Octagon Press.

[26] Philip Tchernavskij. 2019. *Designing and Programming Malleable Software.* Ph.D. Dissertation. Université Paris-Saclay.