Escaping the Prison of Style

Antranig Basman[a] and Philip Tchernavskij[b]

a Raising the Floor, International

b Inclusive Design Research Center, OCAD University

Abstract We follow the received notion of "style" in programming, which is implicitly argued to share many of the values of the quantity of the same name in other areas of human expression, such as literary and artistic expressions. We argue that this is not so, and that this correspondence obscures the essential imbalance in power relations in the ecologies surrounding computational artefacts, and also the fundamentally different nature of these styles in structuring dialogues amongst participants. We consider instead that efforts in cataloguing varieties of "programming styles" merely succeed in capturing variation amongst imprisoned expressions. 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 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.

ACM CCS 2012

- **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

The Art, Science, and Engineering of Programming

Perspective The Art of Programming

Area of Submission Social Coding, General-purpose programming

1 Introduction

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 more incidental details of expression such as formatting, naming and granularity of structure. This narrow sense is mostly followed by **?**] which is a compendium of maxims aimed at producing designs whose code is more easily readable by programmers, and which are more likely to execute more reliably, efficiently and behave better under maintenance.

More ambitious is Lopes’ [**?**] survey of 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. Inevitably, this involves the consideration of entire programming languages which were constructed with the goals of particular programming styles built-in.

Lopes’ survey consists of an evolving set of sample programs written in Python to solve the same task[1] and an accompanying book describing each style, discussing its trade-offs, and suggesting programming exercises, such as comparing different styles, or extending one of the samples with additional behavior [**?**]. 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.

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. The survey illustrates the overlap between notions such as language paradigms, models of computation, and hardware constraints:

> *[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. [**?** , p. xii]*

Next, we discuss what Lopes’ work reveals about programmers’ assumptions about the expressive space of programming, and what kinds of value judgements and distinctions are helpful in navigating that space.

2 These styles delineate the boundary of a prison

We join this workshop’s call to make digital tools more convivial, i.e. making 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.” [**?**] We are concerned by the role existing tools and techniques for creating, maintaining, and using programs have in excluding

[1] At the time of writing, 40 different styles are collected at https://github.com/crista/exercises-in-programming-style

large groups of users from having convivial relationships to their tools [? ?]. 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 constructed to exclude all but its original authors 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 [? , 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. We have labelled aspects of this goal and the means to achieve it as continuing design [?], open authorship [?], and malleable software [?]. 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 notion of programming style is a useful focal point for our critique. We argue that the space of possible styles surveyed by Lopes reveals a sort of disciplinary myopia, in which programming is overly concerned with the work of producing an initial, correct, and efficient program, excluding all other labour and perspectives surrounding its extended life cycle. This constrained space defining the salient dimensions of expression within programming is a prison, isolating programmers and programs from participating in convivial creative relationships.

3 These are not styles

All the programs in [?], from the point of the user, are exactly the same one, not just merely variants. They are invoked in just the same way and produce identical output. In almost every case, the programs are designed in a way that precludes anyone encountering the running program from attempting to understand, change, or integrate it. In this sense, the ecology of function – the network of people and tools surrounding the running programs – is completely cut off from the ecology of design – the corresponding network involved in the creation of the source code [?]. These are "styles" only evident to the elite construction ecology.

Compare this with **?**]'s presentation of style in his architectural design patterns, familiar to programmers through their ripoff by **?**]. 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.* [?]

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. [**?**]*

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). They are variant "approaches to a problem" rather than members of a vocabulary. By contrast, programming styles — and programming design patterns — are used to end conversations through classification, not begin them through inspiration.

4 Like Design Patterns, a raid

In practice, this invocation of the notion of "style" is very similar to the disciplinary raid executed by members of the Design Patterns community [**?**] on Alexander's Pattern Language. In a keynote delivered to OOPSLA shortly thereafter, **?**] decried this moral deframing of pattern language and its lack of community orientation, and described the members of the technology community as behaving as "guns for hire." 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. [**?**]*

5 Like Design Patterns, a catalogue of what must be eliminated

Another similarity with the design patterns ontology is that such a survey of programming style in practice functions as a catalogue of what must be largely eliminated in a move to open, convivial programming. Being inward-looking and technocratic, it is naturally a catalogue of things which must be eliminated or somehow hidden from view once we accept that every participant in the ecology is at least in theory on a common footing, with equal rights to make and receive expressions. By definition, it is a catalogue of the inessential or at best what is secondary.

However, the exhaustive breadth of the survey in [**?**] only shows us the greater extent of the challenge of building a truly open environment. Some modern programming languages (Newspeak [**?**], Infusion [**?**]) are succeeding in making the majority of design patterns (especially constructional ones) obsolete. However, we face the task of designing a user programming system that can make it irrelevant or quietly parameterisable whether a design uses continuations, tail recursion, a publish-subscribe system, sensible defaults or aggressive error-checking, is expressed in terms of pure functions or stateful objects, is expressed in dynamically loadable plugins, or is distributable across huge dataspaces. This task is so large as to seem completely prohibitive, yet it is inevitable that our field, if successful, will tackle it, and it certainly helps to have all our enemies listed in one place.

A precedent for elimination of inessential programming style through parameterisation is found in **?**], in this case invoking a notion of "interface style".

6 There are worse things than a catalogue of styles

So as not to appear backhanded, we should be clear that we greatly welcome the catalogue of program styles assembled in **?**] since it genuinely shows our field at its best. Travel, even travel within a prison, broadens the mind,[2] 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 exceed those of their colleagues. Notable examples include Hoyte's [**?**] declaration "macro programming is, of course, not about style. It is about power", 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. These inhabitants are what **?**] 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"[**?**].

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

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.

7 In part of the prison, there is more light

The survey of styles is presented in the traditional disciplinary framing, which is neutral to all values except those of the discipline, such as efficiency, terseness and correctness. No attempt is made to consider values arising as a result of different communities being put into relation by the programs. In fact, any values originating in the situations where the styles were originally developed are effaced by putting all the “style fragments” onto a common footing as expressed by Python source code invoked from the command line.

For example, style 1, “Good Old Times”, and style 26, “Spreadsheet” originally enjoyed useful virtues of externalisation, i.e. the ability to expose a running program’s behaviour and state in a document form that can be freely exchanged and modified. Externalisation is a key value for living, convivial software, because it supports ongoing (re-)design and integration [**?**].

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.

This need not be so, and not all the style elements are in equally dark areas of the prison. Style 14, “Hollywood”, and Style 15, “Bulletin Board”, recognise in their writeup that they are suited to the design of systems whose evolution cannot fully be foreseen — from page 119:

> *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. [**?** , p. 119]*

However, fuller recognition that these styles might enable the possibility for open communities, which must not require members to rewrite the expressions of other authors in order to use them in designs over which they have full ownership [**?**], 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. [**?** , 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:

```python
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”. Let’s imagine a variant style continuing with this design intention towards open authorship, but stepping back to the simpler functional style 9, “The One”, and incorporating some features from part V of the book on Reflection.

These reflective faculties are seen in three example styles in this chapter, 16, “Introspective”, 17, “Reflective” and 18, “Asides”, but in the first, they are merely used to create obstructions to execution, in the 2nd they are used to further obscure the program’s source text from the ecology of construction, and in the 3rd they implement an extraneous piece of functionality (profiling) only of interest to technicians.

8 Style 15b - “Weak Tea”

This program still lies within the convex hull of our prison, combining elements seen in several different styles surveyed by [?], such as the functional styles 9, “The One” and 24, “Quarantine” and the reflective facilities broadly surveyed in chapters 16, 17 and 18 (although not the same ones in detail), as well as the configuration language style seen in style 19, "No Commitment". 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 [?] 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;”.

This leads us to reconstruct several elements from this previous work in the context of this small ecology of Python programs — firstly a minimal configuration language (or integration language, in the sense of [?]) expressed as a hash of JSON records (similar in construction and intention to parts of Infusion [?]), determining the dataflow and sequencing of the Python functions in *A*’s design. Secondly, a “program addition operator” ⊕ [? , section 5.3] which is capable of fusing *B*’s differential design onto *A*’s, which through the alignment properties of the configuration language can be expressed by a simple dictionary merge. Finally, a minimal selector dialect which allows the topology of the dataflow to be expressed with respect to the naming structure of the configuration language. Since this is a minimal example, more advanced features such as deeply-structured predicate selectors allowing for query-based extension, and deep multiple inheritance hierarchies for configuration are not supported. The resulting "minimal viable integration language" occupies about 50 lines of Python and is not reproduced here,[3] but the program itself, shown in Listing **??** and the corresponding configuration structure in Listing **??**.

Note that listing **??** is extremely similar to the implementation in [**?**] 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.

8.1 About the Configuration Language

The configuration language shown in listing **??** makes use of three forms of selectors, 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 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 5, "Candy Factory", 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. We confess that we looked ahead to the differential design exercise before designing the configuration language, but on the other hand, this full externalisation of computation structure was already mandated by the core principle of avoiding "excess sequential intention" as set out in section 5 of [**?**] and section 2.5 of [**?**].

8.2 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 **??**.

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

[3] 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 configuration 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 **??**

```python
#!/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 **??**

```json
{
    "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 **??**

Interesting features in Listing **??** 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 **??**. 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. However, we note that the addition system is still imprisoned within the

```
        # 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 ⊕

boundaries of the Python ecosystem — we cannot meaningfully combine programs before the relevant module has been referenced via an `import` statement. Section **??** speculates on routes out of the prison that necessarily stray into the operating system's notion of the structure of dynamically loadable objects, and [**?**] describes the "Nexus" system built on top of Infusion that permits program addition to occur via HTTP verbs.

9 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 **??** 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 Listing **??** are largely moral. We have shifted a substantial part of the design out of the reach of arbitrary Turing-complete computation into an impoverished representation that should be easy to author with a variety of structured tools. Also, we've provided an arena in which the expressions of different authors, perhaps expressed using substantially different visual or non-visual representations, can be combined, without privileging any of them as central. 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 **??** as equivalent to its compact functional binding equivalent when considering 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.

9.1 Imagining a Community

Equally damning is imagining a community for which representations such as listing **??** 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 6, "Pipeline". 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.

9.2 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, the authorial flexibility 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. In practice we need a much more ambitious system for accounting for the contents of memory and establishing such correspondences, as sketched in [**? ?**].

9.3 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, index 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 simply 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 [**?**] 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.

9.4 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", 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 salutory 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:

> *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. [**?** , 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.

10 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”[**?**], 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."

11 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.

Acknowledgements The authors are indebted to Colin Clark for illuminating conversations leading to some of the ideas in this paper.

About the authors

Antranig Basman is a software developer, mathematician and technology theorist. He is the owner of an indolent, well-travelled cat. Contact him at amb26@ponder.org.uk.

Philip Tchernavskij is a computer-mediated activity researcher and computer scientist. He is in an inter-employment and -residence limbo. Contact him at philip@tcher.tech.