Ascending the Ladder to Self-Sustainability

Achieving Open Evolution in an Interactive Graphical System

Joel Jakubovic
University of Kent
Canterbury, United Kingdom
jdj9@kent.ac.uk

Tomas Petricek
Charles University
Prague, Czechia
tomas@tomasp.net

Abstract

Programming is usually based on an inconvenient separation between an *implementation* level and a *user* level. *Self-sustaining* systems expose their implementation at their user level so that they can be modified and improved from within. However, the few examples that exist are tightly linked to textual language-based accounts of compiler bootstrapping. If we want systems to be truly open for modification, we need to step beyond programming *languages* and support more structured, visual ways of programming as well. How the bootstrapping process can work in such an interactive context is important yet unexplored territory.

This essay is a critical report on our first-hand experience of building one such system named *BootstrapLab*. We trace and reconstruct the steps for achieving self-sustainability in an interactive, structured, graphical context: choose the platform; design the substrate; implement temporary infrastructure; implement a high-level language; pay off outstanding substrate debt; provide for domain-specific notations.

Throughout, we discuss the challenges involved, identifying design *forces* that shaped the decisions and capturing *heuristics* that resolved these forces in our case. Both positive and negative results are featured, including the efficacy of the heuristics. We close by suggesting how to generalise what worked in our particular case to alternative paths and starting points. The enterprise as a whole takes us a further step towards achieving open and malleable programming systems for everyone.

ACM Reference Format:
Joel Jakubovic and Tomas Petricek. 2022. Ascending the Ladder to Self-Sustainability: Achieving Open Evolution in an Interactive Graphical System. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3563835.3568736

Onward! '22, December 8–10, 2022, Auckland, New Zealand

1 Introduction

In the ordinary lifecycle of software, there is a hard separation between the *product* and its *source*. The product may be any end-user application such as a game, and is created from the source by a *producer*, which is a compiler or other similar tool built for programmers. In this arrangement, the product's user has little ability to re-program it, beyond setting configuration parameters anticipated ahead of time. The user's only option is to modify the source (if it is available) and use the producer to create a new version of the product.

Curiously, this arrangement isn't limited to end-user products but also applies to most *programmer*-oriented products! In the ordinary programming experience, tools like the compiler or editor are themselves products with a separate source and producer. If the user of a language wants to re-program it beyond a customisation anticipated by its designer, they have to go and modify the compiler source code. If they are lucky, the compiler is also written in the same language. In this case, the user is already familiar with the language's notation and capabilities, making the task easier than learning an entirely new language. Even so, their changes still occur at a separate level from their ordinary use of the system.

In this context of programming, the separation between the product and the source is particularly lamentable, as it makes it very hard for programmers to improve their tools[1]. If the language ecosystem is not created using itself, a programmer's mastery of the language is worthless for adapting it. Even if they get lucky as described, the burden of getting the source code, recompiling and deploying it, and maintaining a fork would prevent many from succeeding.

An alternative to this arrangement is *self-sustainable* systems which dissolve the distinction between the product, source, and producer. A single environment provides not only for using and re-programming the *product*, but also for re-programming the *producer*, i.e. the system itself that is used for the programming. These systems are carefully designed to avoid "baking in" any of their behaviour. Instead, they expose as much of their own implementation as possible for modification at the user level. The advantage of such an approach is that innovation and improvement in the system can feed back into its own development.

[1] This is true in a global sense, but even more important in the sense of local adaptations for their own purposes.

For example, consider mathematical notation. It involves many font styles and symbols as infix operators. Yet to express this in code we are limited to ASCII characters and, in many languages, prefix functional notation for custom operators. There are numerous domains where an improvement to this notation would make code easier to follow, such as rendering 3D graphics or computing text layout. If we implement a user interface for entering expressions in mathematical notation, not only would it help us program an end-user application such as a 3D game, but the same notation also becomes instantly available for our own use in-system. We could re-express parts of the code for the system's own user interface, such as its algorithms for text layout. In fact, we could even use our mathematical notation to re-implement the very user interface for the mathematical notation! This would not be the case if the end-user code existed in a different world than the programming system.

This "innovation feedback" encourages a virtuous cycle of improvement regarding notations and beyond. The same can happen when we develop better debugging and testing tools, user interface builders, provenance tracking or performance optimisations. In other words, the development of the system itself will benefit from any improvement made while using the system. This is the key advantage self-sustainable systems have over others.

2 Contributions

The goal of this essay is to lay the foundations for creating new self-sustainable programming systems. The user of such a system should be able to use it for building new products and, along the way, gradually learn how to make increasingly complex improvements to the system itself using domain-appropriate, often graphical, notations.

To do so, we critically analyse the process of bootstrapping a single novel self-sustainable programming system, which we call BootstrapLab, and identify ideas that may apply more generally. This essay presents a rational reconstruction of the steps of the bootstrapping process. For each step, we describe the general task at hand, illustrate this with concrete decisions made in the implementation of BootstrapLab and, where appropriate, sketch possible alternative decisions and their likely consequences. In other words, it is a depth-first exploration of the process with some alternative branches suggested along the way.

The essay can be read at three different levels:

- First, it tells the development story of a concrete system. BootstrapLab is a novel self-sustainable system, based on structured editing, built on top of the web platform.
- Second, it presents a rational reconstruction of the logical steps needed to bootstrap a self-sustainable programming system.
- Third, it highlights design *forces* and *heuristics* for resolving them which can be used by designers of future self-sustainable systems.

We conclude by identifying which parts of our journey ought to be transferrable to other contexts, suggesting a *general technique* for interactively bootstrapping self-sustainable systems from any starting platform.

3 Related Work

The notion of "self-sustainable system" is difficult to discuss purely at the programming language level, because it crucially depends on how program execution and production is interconnected. For this reason, we talk about *programming systems* [7, 13], which include not only programming languages, but also IDEs, programming environments with non-textual notations, and other tools for creating software.

In the context of programming languages, the compiler or interpreter for a language can be implemented in the language itself. This is known as *bootstrapping* and it allows the language creator to co-evolve the language and the compiler. However, they typically have to do so outside of the environment used for building other products. A related concept is that of *meta-circular evaluator*, which is an interpreter for a language, written using the language, that implements features by deferring to its own inmplementation. For example, a meta-circular interpreter for Lisp in Lisp would implement `eval` by calling `eval`. This makes the task of writing the interpreter easier, but it does not eliminate the distinction between the product (application) source code and producer (interpreter) source code.

The two best-known self-sustainable programming systems are Lisp and Smalltalk. Both are typically discussed in programming language terms, but they are more interesting as programming systems. In Smalltalk and many implementations of Lisp (e.g., Interlisp), the system itself (producer) can be modified from the same environment that is used for creating products. In other words, a Smalltalk image contains both the objects that make up the product as well as the objects that make up the Smalltalk environment itself. We also regard Unix as a whole to be a self-sustainable system, though the individual programs within it seldom have this property.

Most literature discussing self-sustainability [11, 12] seems to focus on textual languages as the way to get there. We broaden the scope to programming systems, because this is necessary in order to talk about interaction and graphical capabilities. We desire to support graphical or structured ways of expressing programs that go beyond text [4, 6, 9], and feel that this has been neglected in prior work.

The one system that directly influenced our work is the Combined Object Lambda Architecture or COLA [18]. The system is described as a mutually self-implementing pair of abstractions: a structural object model (the "Object" in

the acronym) and a behavioural Lisp-like language (the "Lambda"). The system aims for maximal openness to modification, down to the basic semantics of object messaging and Lisp expressions. The self-sustainability allows for innovation feedback that the authors refer to as *internal evolution*:

> Applying [internal evolution] locally provides scoped, domain-specific languages in which to express arbitrarily small parts of an application (these might be better called *mood-specific* languages). Implementing new syntax and semantics should be (and is) as simple as defining a new function or macro in a traditional language.

The aim of the COLA design is to create a maximally flexible self-sustaining system, but its exposition has three limitations:

1. It restricts the form of notations to text, curtailing the ambition of pervasive domain-specific adaptation. We would prefer mood-specific *notations* or *interfaces* generally, including *but not limited to* text.
2. COLA's support of programming languages is presented as a pipeline of traditional batch-mode transformations such as parsing, analysis, and code generation. This further steeps it in a world of linear sequence transformations that obscure the interesting ideas.
3. The bootstrapping process of implementing such a design is also cast in terms of batch-mode transformations of various source code files. We would rather have the ability to gradually *sculpt* a system into a self-sustainable state, interactively, through a combination of manual actions and automatic code.

Because COLA was such a big influence on the present work, we will refer to it repeatedly for comparison and offer some analysis of our own.

4 Design Objectives

We follow in the spirit of COLA, but we aim to bootstrap a graphical and interactive self-sustainable system instead of a textual one based on batch transformations.

We want to support not just textual domain-specific languages, but visual domain-specific notations. A user of the system should be able to express their thoughts in a diagrammatic form if they so wish. Support for graphics should be built-in in the system. In short, we desire *notational freedom.*

We also want to work with an *interactive* system. The user should be able to modify the state of the running system through manual gestures, not just programmatically.

This approach can better exploit the graphical and interactive capabilities of modern computing, but it also sidesteps the tedious accidental complexities of parsing and serialising text. Similarly, making the system interactive will allow the user to better understand the consequences of individual small changes and will, in turn, support the virtuous cycle of self-improvement.

In technical terms, we do not write an initial object system in a language like C++. Instead, we choose as our starting point a *platform* equipped with an interactive REPL as a "blank slate". We then gradually (in the ideal case) "sculpt" this into a self-sustainable state.

Our desire is to make an interactive, structured "port" of the COLA approach. This is unexplored territory. It must be emphasised that finding the right "final design" upfront should not be necessary and would, in fact, defeat the spirit of the enterprise. The point is to build an initial kernel which is then sufficient for evolving and improving itself.

In order to support interactivity, structure, and graphics, a natural place to start is a non-self-sustainable implementation platform that already conveniently supports those features. This will make it possible to start with a suitable "blank slate" and, gradually, develop the system into a self-sustainable one. At each stage, we take stock of what changes can feasibly be achieved at the "user" level within the system, versus those that can only be achieved at the implementation level. We then ask ourselves: how can we imbue the user level with control over some of these aspects?

The following sections propose key steps for evolving self-sustainability in this way, informed by our actual experience applying them in *BootstrapLab*. We will examine the forces and heuristics that motivated these steps, and reflect on their efficacy in light of actual practice.

4.1 Concepts and Terminology

To *bootstrap* a programming language is to carry out the following sequence of steps (more or less):

1. We design a novel language *NovLang* that we want to use.
2. We write *NovLang* source code that will compile other *NovLang* source code into a runnable program.
3. We hand-translate this to C/C++ and build a compiler for *NovLang*.
4. We run this to compile the *NovLang* source code from step 2.
5. We obtain a runnable compiler for *NovLang*, which was written in *NovLang* and is now "self-hosting".
6. We can now discard the C/C++ code.

Our task is to explore the question: how do we bootstrap *interactive graphical systems?*

In this essay, we use the term *product system* (or simply "the system") to refer to the programming system that we evolve towards self-sustainability. We use the *producer system* (or simply "producer") to bootstrap the product system. The producer is divided into two layers: the *platform* consists of all the pre-existing capabilities of the producer, while the *substrate* is the basis for the product system that we have to build. We use the term *in-system* to refer to changes made within the product system, by using it as a programming system at its user level.

We model the product system as having a *state* that *changes over time*. This is necessarily the case for any interesting interactive programming system, regardless of its programming paradigm. Even in functional, declarative, reactive or logic paradigms, the evaluation or re-computation in response to interaction with a user produces a new (changed) state.

When discussing state, we refer to both the visible interface and the hidden internal state of the programming system. The state always consists of substructures such as byte arrays, object graphs or trees. Correspondingly, a change to the state can be decomposed into sub-changes that affect small parts of the state.[2] This gives rise to primitive *instructions* that describe state change at the finest level of granularity.

By definition, what we do with a self-sustainable system is open-ended. This essay is solely concerned with *getting there* from the ordinary world, which is why we spend so much discussion on the design of the substrate. This determines how the product system can evolve, how soon can it become self-sustainable and to what extent. The design is shaped by *forces* that we aim to make explicit. We consider ways of resolving competing forces and highlight these as *heuristics* throughout the essay.

5 Journey Itinerary

The rest of the essay documents the steps involved in designing a self-sustainable system. Be advised that the sequence is a *rational reconstruction*. The implementation of BootstrapLab followed a more meandering path, but the following steps gesture at the Platonically optimal pathway for bootstrapping a self-sustainable programming system:

1. **Choose a starting platform.** The platform is a *pre-existing* programming system that we use to create and run the product system. The platform cannot be re-programmed, let alone to become self-sustainable, but it allows us to build a self-sustainable product system. To choose a platform, we consider its distance from desired substrate features and personal preference.
2. **Design a substrate.** The substrate defines the basic infrastructure supporting the product system: how the state is *represented* and *changed*. The design of a substrate re-uses parts of the platform where possible and extends it where necessary. We must decide which platform capabilities to use to represent the state and how to expose graphics and interaction. We design a minimal *instruction set* describing changes to the state, which can be represented using the state. We then use the platform to implement an engine that executes these instructions.
3. **Implement temporary infrastructure.** Use the platform to implement tools for working within the substrate, most importantly a *state viewer* or editor. These tools constitute the "ladder" that we will pull up behind us once we have ascended to in-system implementations of these tools.
4. **Implement a high-level language.** The substrate's instruction set (ASM) is cumbersome, so ensure programs can be expressed in-system via high-level constructs. Decide how to represent expressions as structured state and whether to *interpret* or *compile* them into ASM. Ideally, develop such an engine in ASM gradually and interactively. Alternatively, implement it at the platform level and later *port* it to ASM or the high-level language itself.
5. **Pay off outstanding substrate debt.** Port all remaining temporary infrastructure into the system, taking advantage of the infrastructure itself and the high-level language. The result is a *self-sustainable* programming system.
6. **Provide for domain-specific notations.** Use the self-sustaining state editor to construct a more convenient interface for editing high-level expressions. Add *novel notations and interfaces* as needed. Use these not just for programming new end-user applications, but also to improve the product system itself.

What we have here looks like a Waterfall development plan, each step strictly following from the completion of the previous. This presentation is convenient as a summary, but in practice, the sequencing here need not be so rigid. Adjacent steps may overlap, or we may need to prototype and revise a previous step in light of the result.

The general outline also resembles the (discredited) "recapitulation theory" in biology. The idea was, in order for an embryo to develop into a full organism, it passes through the evolutionary history of its ancestors. In other words, for a *particular* clump of cells to develop into an animal, it needs to fast-track its ancestors' evolution from a small clump of cells in the distant past.

While this has since been rejected in biology, it is a good summary of what is going on in our project here. The bootstrapping of a particular self-sustainable system fast-tracks the historical development of computing's abstractions. It begins at the machine level and ascends through to higher-level languages, each time building the next stage in the current one. Our work can be seen as an attempt to reconstruct programming on top of a more structured, graphical substrate than the byte arrays we all had to use the first time around. With that in mind, let us now proceed to the first stage.

6 Choose a Starting Platform

The platform is a *pre-existing* programming system that we use to create and run the product

[2] It may be argued that a very high-level programming paradigm would make it impossible to affect only small parts of the state. This may, however, not be a suitable starting point for a self-sustainable system.

system. The platform cannot be re-programmed, let alone to become self-sustainable, but it allows us to bootstrap a self-sustainable product system. To choose a platform, we consider its distance from desired substrate features and personal preference.

The first step is to choose the platform that we will use as the basis for the product system. This could be any existing high-level or low-level programming system. One key factor is simply personal familiarity or preference for a particular platform. This plays a role during bootstrapping, but is destined to become irrelevant once self-sustainability is achieved.

The other consideration is the primitives provided by the platform. They influence how we can design the substrate on top of it in the next step. If we begin with a high-level platform with many convenient features (or, say, graphics or audio capabilities), then we will have to regard them as black boxes. We may expose them as primitives in the product system, but we will not be able to *re-program* them in-system since we cannot re-program the platform. Alternatively, such imported convenient high-level features could later be *re-implemented* in the product using more basic primitives. This would, however, delay the point from which we can work fully in-system. This foreshadows a coming design tension in the substrate (Section 7.2).

In early computing, such as the Altair 8800, the platform was linear memory (state) and native CPU instructions (state change). The platform did not provide other tooling aside from switches to manually set memory values.

In COLA, the platform is C [19] or C++ [18] and the Unix command-line environment; in other words, it is the Unix programming system [13].

In BootstrapLab, we chose JavaScript and the Web platform generally. This provides us with built-in Web technologies and libraries (including graphics) and the browser developer tools. This platform provides a range of convenient tools to assist bootstrapping. Because of its large scope, we have to carefully choose primitives to expose to the product system.

What can be changed at the user level? At this point, there is no product system to speak of yet. This means that nothing can be changed in-system. The platform can, in principle, be modified, but we assume this is so unfamiliar or uneconomical that the reader has opted to make a self-sustainable system instead!

7 Design a Substrate

The substrate defines the basic infrastructure supporting the product system: how the state is *represented* and *changed*. The design of a substrate re-uses parts of the platform where possible and extends it where necessary. We must decide which platform capabilities to use to represent the state and how to expose graphics and interaction. We design a minimal *instruction set* describing changes to the state, which can be represented using the state. We then use the platform to implement an engine that executes these instructions.

Table 1. The conceptual divisions of the substrate.

Domain \ Agent	Human (Manual)	Computer (Automatic)
State	User Interface	Data structures
Change	UI Controls	Instructions

With the *platform* defined as the already-existing programming system that we start from, we define the *substrate* as the basic infrastructure, implemented via the platform, necessary for the product system. This substrate is the part of the system which we have control over (being programmed *by us*, unlike the platform itself) yet which we do not expect to expose from within the system. In other words, the substrate is the small non-self-sustainable core that supports the self-sustainable product on top of it.

In short, our division is as follows:

	Created by us?	Self-sustainable?
Platform	No	No
Substrate	Yes, atop Platform	No
Product	Yes, atop Substrate	Yes

The design of the substrate can be considered along two dimensions (Figure 1). The first dimension follows the distinction between data and code, or *state* and state *change*. We must decide how the *state* of the system will be represented. Often, this is a matter of choosing an appropriate subset of what the platform already provides. Then, we decide how primitive *changes* to that state can be described and define the instruction set.

The second dimension follows the division between the *computer* and *human* actors. The full state of the system will be an internal data structure, but a *part* of the state—comprising the state of the user interface—can be directly seen by the user. Similarly, *change* can be performed automatically or manually. There must be a way to run instructions automatically at a high speed, but the user interface must also provide controls for a human to make changes at their own pace.

While the foregoing model applies to programming systems generally, a special condition is required for those that are self-sustainable. We must represent instructions as pieces of state, as opposed to having "two types of things"—ordinary data, and code—which must be viewed and edited using completely different tools. This property, conventionally known

as *homoiconicity*, means instructions can be generated and manipulated just like ordinary state, whether programmatically or manually. Only if this is possible can higher-level abstractions can be built up, in-system, from the low level.

Requirement 1 (Homoiconicity)**.** Instructions must be readable and writeable as ordinary state.

7.1 COLA's Low-Level Byte Arrays

In COLA, the substrate is quite minimal and the majority is inherited "for free" from the low-level runtime environment provided by Unix.

At the lowest level, state in COLA consists of an array of bytes, addressed numerically. Some structure is imposed on this via C's standard memory allocation routines, refining the model of state to a graph of fixed-size memory blocks (plus the stack). Changes to this state are represented as machine instructions encoded as bytes. This is the basic state model of a C program; the sample code for COLA embellishes this with little more than a way to associate objects to their vtables[3], and a cache for method lookups.

This bare-bones, low-level substrate does not require much development on top of the platform and so it is quicker to complete. The ontology of state is copied from the platform, and in this case the machine instructions can be inherited too.[4] Completing the substrate more quickly means we can start working in-system sooner, but there is a downside: it may be cumbersome to work with such minimal functionality. The unfortunate effect would be that we speed through a primitive substrate, only to suffer slow progress at the beginning of in-system development.

Building back up from machine-code level may be an impressive hacker achievement or useful for pedagogy [1]. But it is clearly not optimal, speed-wise, when we already have a higher-level platform to program with.

In the other direction, there is no limit to how fancy we could make the substrate in terms of high-level abstractions and convenient features. However, these would take much longer to implement and delay in-system development. Moreover, this risks doing a lot of work that can never benefit from in-system innovation, because the substrate's implementation will not be modifiable from within the system it supports.

7.2 The Major Design Conflict

We clearly have two opposing tendencies here, which we will formalise as follows:

Force 1 (Avoid Boilerplate)**.** Push complex features into the substrate to avoid wasting in-system development time on them.

Force 2 (Escape The Platform)**.** Push complex features in-system to avoid delaying in-system development and to have them benefit from in-system innovation.

We will refer to these throughout the journey. They conflict over where the implementation of convenient functionality should reside. In any specific design, they will resolve in some compromise. It is helpful to consider the extreme points of this.

Force 2 wants to get the substrate over with as quickly as possible, eager to escape the (real) limitations of the platform and get working in a system that *can* be arbitrarily changed. Force 2, left unchecked, will guide us to adopt a substrate resembling a Turing machine: have a tape for the state; instructions for manually shifting left and right, reacting to the current symbol, and writing a new one; have a user interface in which to do these things manually. Such a substrate is so simple it could be coded in an hour or two. Yet our first duties in-system will be to implement extremely basic features, like data addressing and arithmetic, in an extremely tedious way. The endpoint of Force 2 is the Turing Tarpit.

On the other hand, if we follow Force 1 unchecked, we spend much time and effort working with the platform to produce, in effect, a *complete* novel programming system. Any feature we would find useful in-system, we would have already implemented outside it. Yet this means that all the important functionality could not be changed except by going back to the source code in the platform; we'd have created a boring old *non*-self-sustainable programming system. The endpoint of Force 1 is programming-as-usual.

A symptom of the latter failure mode would be that we never felt comfortable leaving the platform behind and continuing development from within the system. Self-sustaining systems are meant to be *grown* from a small enough starting point; we shouldn't need to come up with a flawless design ahead of time. This will only be possible if we artfully balance Forces 1 and 2 so that the in-system programming experience becomes tolerable in a reasonable timeframe.

7.2.1 Reflections on the Bare-Bones Approach.

We experienced something like the Force 2 absurdity for COLA when following the sample C implementation of its object system [19]. The code was easy enough to comprehend and compile, but what we were left with was a system living entirely in memory lacking even a command-line interface. In order to develop the system, it seemed necessary to run it in a machine-level debugger.

Even so, we would still be stuck in the low-level binary world which is unfriendly for humans to work with. Instead of names, we only have numbers for addressing things. Additionally, the state is *flat*. We cannot insert or grow something

[3] A vtable specifies object behaviour by supplying runnable code for a requested method name. It is separate from the object "instance" so that multiple objects can share the same behaviour.

[4] In general, the internal representation of code in the platform will be unavailable to us when programming with it, so we expect not to be able to inherit it. This low-level platform is a special case, where we do have access to code if we are willing to write instructions using their numerical codes.

without physically moving other content to make space. Any structure, such as trees or graphs, has to be *faked* as memory blocks pointing to each other.

This type of substrate is still better than a Turing machine, and was a historical necessity in the early days of computing. But nowadays, we have the opportunity to leave this behind, and instead build new systems on top of a "low level" that is nevertheless *minimally* human-friendly.

Heuristic 1 (Minimally human-friendly low-level)**.** Ensure the substrate natively supports *string names* and *substructures*. This is a minimal response to Force 1 that still keeps the substrate simple enough and thus does not strongly conflict with Force 2.

7.3 BootstrapLab's Simple, Structured State Model

For the design of BootstrapLab, we choose the Web platform and JavaScript for personal preference reasons. This imposed a number of design decisions on the substrate, due to a tendency for earlier choices to determine which later ones will feel "natural" or "fitting".

In our high-level platform language JavaScript, state is a graph of plain JavaScript objects acting as property dictionaries. Suppose we *still* chose a low-level binary substrate like that of COLA. This would no doubt be possible: declare one giant JavaScript array called `state`, design numerical instruction encodings which overwrite numbers at certain indexes, etc. Yet this would feel like a perverse waste of something the platform was giving us for free.

JavaScript already provides the basic human affordances of naming and substructure, so why would we throw them away and force ourselves to implement them in-system? The low-level COLA substrate does plausibly follow from its base C platform. Our choice of JavaScript as the platform encourages us to preserve its own state model in the substrate we design.

Similarly, it would make no sense to represent instructions as numbers or strings. While in the binary world, machine instructions are byte sequences with bitfields for opcodes and operands, in a dictionary substrate inherited from JavaScript, it makes sense to have explicit fields for this data:

```
{
  operation: 'copy',
  from: [ alice, 'age' ],
  to: [ bob, 'age' ]
}
```

As this example shows, addresses in a dictionary-based state model consist of an object reference and a key name.

This "preservation" incentive pervades the journey from platform to product system. The substrate should leverage representations made possible by the platform, while the instruction representation should leverage the structuring of state provided by the substrate. This will also apply to further subdomains expressed in the state model, such as graphics and high-level programming expressions. We formalise this as the following:

Force 3 (Alignment)**.** Everything should fit: instructions, high-level expressions, and graphics expressions should all fit the substrate, and the substrate should fit the platform.

In the end, our substrate largely inherits the state model, only making simplifications. For example, JavaScript objects have prototypal inheritance, meaning that a simple "read" operation of a property requires potentially traversing a chain of objects. Our substrate here omits this, so reads are quite simple. Additionally, JavaScript includes a special `Array` object type. We omitted this, opting to represent lists as maps[5] with numerical keys. This unification means that the state model only has one type of composite entity, a fact we will exploit later for the high-level language.

We also noticed that we would not get very far if all our progress in-system could be wiped clean by losing our browser tab. Our platform does not provide *persistence* out of the box, so we had to implement a mechanism in the substrate. We walk the state graph from the root node and discover a spanning tree, specially marking cyclic or double-parent references. We then serialise this into a JSON file which we can load by undoing the process. This is reminiscent of the image-based persistence in Smalltalk, though it is frustratingly manual. Nevertheless, it was critical to patch this unfortunate aspect of the platform and this was enough to do so.

Even though JavaScript is a high-level language, we consider this substrate low-level *relative* to the platform below it. Force 1 gave us several ideas for convenient features of a smarter state model, but Force 2 urged us to press ahead without them and see if we needed them later; see Appendix A for details.

7.3.1 Designing the Instruction Set.

While the "data" half of the substrate may be easy to inherit from the platform, the "code" half is typically not. Simply including an interpreter for source code in JavaScript is not an option, as this would embed a reliance on a strings and parsing in the core of the system.

Slightly better would be an interpreter for the JavaScript abstract syntax tree. However, Force 2 still pushes against this. A high-level language interpreter is nontrivial even without parsing and would delay our ability to work in-system. Also, an interpreter is a computer program; this program, or parts of it, might be best expressed or debugged via particular notations; by having it in the substrate, we'd restrict ourselves to the interface of JavaScript in our text editor.

Instead, consider what it takes to implement the interpreter for Assembler, a.k.a. the Fetch-Decode-Execute cycle.

[5] We refer to our substrate's basic dictionary structure as the *map* for brevity.

We fetch the next small change to make to the state (an instruction). We do a simple case-split on the opcode field; we carry out some small change to the state; rinse and repeat. With this, we can surely mirror the real-world development of higher-level languages from lower ones.

Heuristic 2 (Use Imperative Assembler)**.** Begin from imperative assembler, as this allows us to make arbitrary changes to the state using a minimal interpreter that is quick to implement. Force 2 outweighs Force 1 here.

It is important to stress that this "assembler" is relative to the form of the substrate. If the substrate is binary memory, "assembler" will refer to machine instructions. But in our case of a minimally human-friendly low level (Heuristic 1), there is nothing binary about them. The instructions express operations on structured objects with names and are, themselves, represented as structured objects with names. Similarly, "imperative" just refers to the fact that the instructions are arranged in a sequence from the point of view of the interpreter, because it is easier to implement a fetch-execute cycle than, say, a resolver for a dependency graph. The above considerations lead us to Heuristic 3.

Heuristic 3 (Simple Assembler)**.** Prefer fewer instruction types (RISC) over more (CISC). This reduces the size of the interpreter and will be quickest to implement. It will make programs longer, but this can be mitigated by a high-level programming language. Force 2 outweighs Force 1 here too.

Right away, we know there will have to be a special piece of state for the *instruction pointer*. This could indicate the *current* instruction or the *next*; we chose the latter for BootstrapLab and called it `next_instruction`.

The value of this "register" is determined by how exactly we fetch the next instruction. Perhaps each one has a `next` field which we can simply follow. In this case, the `next_instruction` will simply be the instruction itself. This also gives us convenient conditionals (e.g. fields called `if_true` and `if_false`) but means that instruction *sequences* will have a nesting structure. This latter consequence may be inconvenient for presentation in a tree view. For BootstrapLab, we chose the alternative of numerically indexed lists of instructions which easily display in a column. This choice determined `next_instruction` to instead hold an *address* made of container map and key name:

```
next_instruction: {
  map: <instruction list>,
  key: 1 // i.e. first instruction in the list
}
```

Here, the "fetch" step involves dereferencing the address and then incrementing the key name.

Next, we turn to what types of instructions we need. Alignment (Force 3) means that, given a state model, obtaining an instruction set should be more of a "derivation" than a hard design problem. This is because some choices are obviously inappropriate and others clearly fitting to the state model. For example, in a tree-structured state model, it would be foolish to have instructions that can only see the root level:

```
{op: 'copy', from: 'source_key', to: 'dest_key'}
```

Without the ability to read or write keys *within* an arbitrary tree node, as far as programmatic change is concerned, the state becomes a *de facto* flat list instead of a tree. Therefore, it is critical that anywhere in the state can be accessed or modified by an appropriate instruction sequence.

The checklist of basic functions for an instruction set to be Turing-complete is as follows:

1. Copy from one location to another (a "literal" is just copied from the instruction itself)
2. Treat a value as an address and follow the reference
3. Unconditional jump (copy a value to the instruction pointer)
4. Conditional jump (take a path based on a runtime condition)

However, Force 1 may push us to include basic boilerplate like arithmetic or an operand stack. Furthermore, it is advisable to have an "escape hatch" into the platform if possible. In BootstrapLab, our platform language JavaScript provides the `eval()` function to execute a string of JavaScript code. We exposed this as a `js` instruction. This allows us to use and store JavaScript code in the *running* system instead of having to edit the source file.

The resulting instruction set for BootstrapLab was derived from these considerations, as well as extreme application of Heuristic 3. It uses the top-level map as a set of "registers" whose contents are *immediately* accessible. State that is "further away" is accessed by following key paths from there, or from existing map references. There are several special registers used by instructions, but other names in the top-level are available as local variables in user code. The instructions are as follows:

- `load` fills the `focus` register with a literal value.
- `deref` treats the value in `focus` as naming a register and copies the register's value into `focus`.
- `index` expects a map in the `map` register and a key name in `focus`. It looks up this key in `map` and replaces `map` with the value.
- `store` copies the value in `focus` to a named register. Alternatively, if no register is present, it copies the value in the `source` register to the destination identified by `map` and `focus`, as with the `index` instruction.

An instruction is represented as a map with an `op` field for its name and other fields for parameters. For example:

```
{ op: 'store', register: 'source' }
```

It is remarkable that these few operations really are sufficent even for conditional and unconditional jumps. A jump is achieved by overwriting `next_instruction`, and this can be conditionalised by `index`ing a map of code paths based

on a selector. We made the decision that `index`, if accessing a key not present in the map, will try and retrieve the special key _ instead. This supports a generic "else" or "otherwise" clause for conditionals.

The minimal, microcode-like instruction set here was an experiment in extreme parsimony; see Appendix C for the gory details. Although it was interesting, certain basic operations (such as jumps) are extremely verbose, taking many instructions. Although it was quick to implement these instructions in JavaScript, it was too tedious to work with them in-system. In retrospect, it looks like we went too far with Force 2 here and fell into its associated Turing Tarpit trap. We thus consider an *extreme* interpretation of Heuristic 3 refuted for the purposes of working in-system sooner. We recommend achieving a better balance by including direct path arguments in instructions (e.g. "copy `a.b.c` to `x.y.z`" as a single instruction), as well as separate (un)conditional jump instructions.

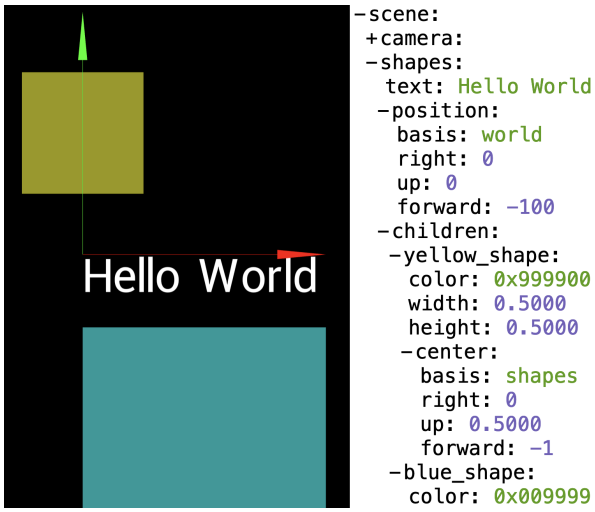

Figure 1. Example of how nested tree fields are represented (right) vs. the rendered output (left). The right-hand half is the temporary state view discussed in Section 8.

7.3.2 Graphics and Interaction. Now we've covered the computer-oriented part of the substrate, we turn to the human-oriented user interface state and change aspects. One way we wish to distinguish BootstrapLab from the related work is that graphical interfaces are present from the beginning and not just an afterthought. There are two factors here: how graphics are represented in the substrate, and how they are actually displayed.

It may be useful to see this as a microcosm of the entire journey. First we must select a graphics library available for our platform (i.e. the *graphics platform*). Then we must decide how graphics are represented in our substrate (a graphics *sub*-substrate) and how these graphics actually end up on our screen.

In BootstrapLab, we chose to build off the THREE.js 3D graphics library as our platform. As for the substrate, we face an immediate choice between so-called "immediate mode" and "retained mode" conventions. In immediate mode, we draw and change graphics by issuing commands; a "code-like" approach. In retained mode, the state of the scene is represented as some structured arrangement of state. When we want to change something, we simply change the relevant part of the state and the display should automatically update.

Immediate-mode in this case could be realised by, say, exposing all the relevant THREE.js functions as special instruction types. In actuality, however, this sounded far too tedious to work with; Force 1 won out and we opted for retained mode instead. The rest of the design then fell out via Alignment (Force 3).

Consider the "flat bytes" substrate in which microcomputer games were programmed. In this world there is a special region of memory, the *framebuffer*, which is treated as the ground truth of pixels displaying on the screen. To draw, programs rasterise shapes into pixels and write to the framebuffer.[6] The framebuffer has a flat structure—two-dimensional, yet not by any means nested—aligning with the substrate it sits within. A natural choice for retained-mode graphics representation can be found by inspecting the substrate. In BootstrapLab's case, a natural choice is not a flat "framebuffer" but a tree structure of data describing shapes and text—vector graphics.[7]

Heuristic 4 (In-state graphics)**.** Make graphical interfaces expressible as ordinary state in a special location. Having graphics built into the substrate responds to Force 1 while Force 3 directs us to use a representation that fits the state model.

In BootstrapLab, this is a sub*tree* of the state under the top-level name `scene` (Figure 1). There are several special keys (e.g. `text`, `position`, `color`, `children`) which have graphical consequences. Other keys may be used as ordinary state.

For interaction, we need to expose the platform's ability to listen for user input. In BootstrapLab, we execute a named code sequence in the substrate from JavaScript event handlers, which now function as "device drivers" (Figure 2).

This is a basic sketch with some issues elided that a complete account would cover. For example, what happens when an input event occurs during the handling of a previous event? Possibilities include ignoring the extra event or providing some sort of stack analogue[8] for nested handlers. Such a data structure may also be necessary for saving and

[6] In Commodore 64 BASIC, this would be accomplished with commands like `POKE 1024,1`.
[7] Further rationale for this approach can be seen in [8].
[8] Of course, in a structured substrate, there is room for improvement on the linear form of the low-level machine stack.

```
window.onkeydown = e => {
  state.set('input', 'type', 'keydown');
  state.set('input', 'key', e.key);
  let input_handler_code = state.get('input', 'handler');
  save_context();
  state.set('next_instruction', new state.Map({
    map: input_handler_code, key: 1
  }));
  asm.execute_till_completion();
  restore_context();
};
```

Figure 2. "Device driver" triggering a generic event handler sequence in-system.

restoring the instruction pointer along with other context. These concerns have analogues in interrupt handling for operating system design, which could be consulted for further guidance.

7.3.3 BootstrapLab Substrate Summary.

Computer state is a graph of maps; lists are just maps with numerical keys. Instructions are `load`, `deref`, `store`, `index`, `js`. Special top-level keys are `focus`, `map`, `source` and `next_instruction`. User Interface state is controlled via the special `scene` subtree of state. Each node may use special keys like `text`, `width`, `height`, `color`, `position`, and `children`, as well as abritrary other keys for user data.

What can be changed at the user level? System state can be modified and instructions can be executed, but only using the cumbersome capabilities of the platform. In case of BootstrapLab, this means using the JavaScript debugging console to edit state and calling a function to execute a certain number of instructions.

8 Implement Temporary Infrastructure

> Use the platform to implement tools for working within the substrate, most importantly a *state viewer* or editor. These tools constitute the "ladder" that we will pull up behind us once we have ascended to in-system implementations of these tools.

In most cases, the base platform will provide some way of viewing and modifying state, but this is typically inconvenient to use. The next step in bootstrapping a self-sustainable system involves implementing temporary infrastructure that lets us work with state more conveniently.

8.1 Early Computing and COLA

Temporary infrastructure to support in-system development can be found in many developments of self-sustainable systems. An example from the early history of computing is the Teletype loader for the Altair 8800. Here, the base platform was the Altair hardware with its memory and native CPU

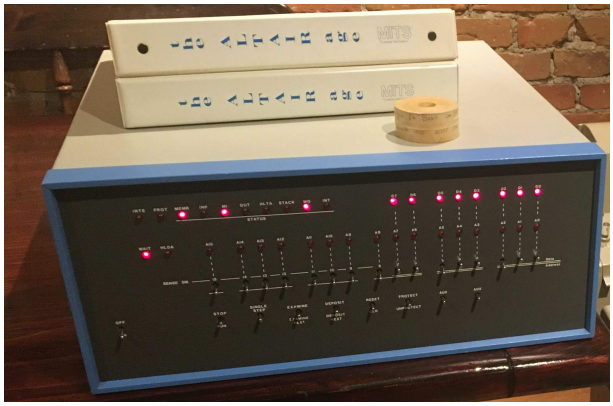

Figure 3. The Altair 8800 microcomputer and its front panel of switches. *Image credit: [5].*

instructions. The only way to modify state through the platform was through the use of hardware switches at the front of the computer (Figure 3), which could be used to read and set values in a given range of memory.

Programming *in-system* looks like the tedious setting of switches to poke numerical instructions to memory. To make entering programs easier, the recommended first step when using the Altair 8800 was to manually input instructions for a *boot loader* that communicated over the serial port. When finished, this could be run to load instructions from a paper tape. From here, programmers could write instructions more conveniently using a Teletype terminal and have them loaded into the Altair memory.

In the COLA architecture [18], there is a four-step process (Section 6.1, Bootstrapping), the first three of which appear to be throwaway. This includes a compiler for their state model in C++. This is aptly "jettisoned without remorse" once it has been re-implemented in itself, though it is unclear how a state model can perform computation (only after this do they implement the "behavioural layer"). Regardless, this clearly echoes the bootstrapping process for programming languages (Section 4.1).

The problem with these steps is that they are hard to port to a context involving structured, graphical notation and interactive system evolution. Our task is to get the system into a state where the platform, in a sense, can be "jettisoned" in terms of our attention, even though the platform-implemented substrate will be running in the background.

8.2 Temporary Infrastructure in BootstrapLab

On its own, our chosen platform for BootstrapLab only has one way to view parts of the state: issuing JavaScript commands via the developer console to poll a current value. This is almost as tedious as toggling switches on the Altair. Being able to see a live view of all of the state would be a highly useful facility early on. In this case, Force 1 wins relative to Force 2 (captured by Heuristic 5) and we implemented a tree

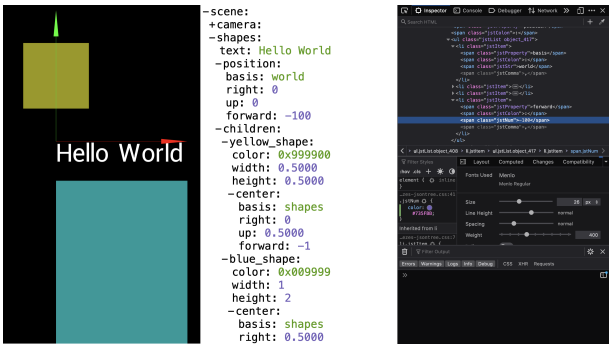

Figure 4. The full BootstrapLab interface. From the left: graphics window, temporary HTML state viewer, and browser developer tools.

view in the substrate based on an existing JavaScript library. State editing can continue to be done via the console (see Figure 4).

The JavaScript tree view is a complex set of functionality set to work and display in one specific way, and all control over this resides at the platform level. The infrastructure cannot be modified from within the system. Therefore, we regard this situation as *temporary*. It is a ladder that we climb to end up in a state where we can implement a (better) state editor in-system. Once a suitable in-system editor exists, we can pull up the ladder (or if you like, "jettison it without remorse").

At this point of the bootstrapping process, BootstrapLab's interface consists of three sections (Figure 4). On the right, we have the browser console, inherited through from the platform's interface. In the middle, we have the output of the platform's main graphics technology, the Document Object Model (DOM), displaying the temporary state viewer. Because we have not chosen to expose DOM control from within the system, the system only affects this area indirectly through ordinary state changes. Finally, on the left, we have the THREE.js-backed graphics window, where we will later build a state editor whose behaviour (including appearance) *will* be controlled from within the system.

Ideally, we would have actually supported interactive state *editing* in the temporary infrastructure, not just viewing. In our case, however, we accepted state editing through console commands until implementing a state editor in terms of the left-hand graphics window (see Section 10).

Another example of temporary infrastructure is zoom-and-pan in the graphics window. Working within a small box is very restrictive if we want to use it for viewing and editing the entirety of the system state. The finite region can be opened up into an infinite workspace by adding the ability to pan and zoom the camera with the mouse. This was important to have early on for BootstrapLab, so once again Force 1 overrode Force 2 and we implemented this in JavaScript.

What can be changed at the user level? The basic activities of viewing or editing state should be made easier by the temporary infrastructure. For the Altair 8800, instruction entry was improved. For COLA, the basic state model was made available in the first place. For BootstrapLab, we targeted state visibility.

Heuristic 5 (Platform editor)**.** As soon as possible, use the platform to implement a temporary state viewer and/or editor. This temporary infrastructure will later be discarded, but given a capable enough platform, it is very easy to implement. For this reason, it is valuable for simplifying further in-system development. Here, again, Force 1 outweighs Force 2.

9 Implement a High-Level Language

> The substrate's instruction set (ASM) is cumbersome, so ensure programs can be expressed in-system via high-level constructs. Decide how to represent expressions as structured state and whether to *interpret* or *compile* them into ASM. Ideally, develop such an engine in ASM gradually and interactively. Alternatively, implement it at the platform level and later *port* it to ASM or the high-level language itself.

The temporary infrastructure created in the preceding step may be enough to allow limited development in-system. However, it does not yet provide the barely tolerable programming experience we would need in order to feel comfortable ditching the platform. For this, an additional step is needed.

To make programming in-system pleasant enough, we need a high-level programming language that executes on top of the system substrate. This means that programs and all their necessary runtime state will be stored in the system state and the execution will be done either by a compiler to the substrate's instruction set or an interpreter.

9.1 Shortcuts for Low-Level Substrates

For a programming system built atop a limited platform (e.g. hardware), the temporary infrastructure may be the best tool that is available for programming. In that case, we would write the compiler or interpreter directly using the instruction set. However, as long as the platform has higher capabilities or one has access to alternative platforms, this may not be optimal. When Paul Allen and Bill Gates wrote the famous BASIC programming language for the Altair 8800, they did not do this *on* the Altair 8800, but using an Intel 8080 CPU emulator written and running on Harvard's PDP-10. The high-level language for Altair 8800 was thus developed *outside the system.*

In COLA, it is unclear how the Lisp-like programming language is built beyond the broad outlines. What is clear is that the bootstrapping process is carried out by means

Lisp:

```
(define fac
  (lambda (n)
    (if (= n 0)
      1
      (* n (fac (decr n)))))
(fac 3)
```

Masp:

```
apply: define,  name: fac,  as:
  apply: lambda,  arg_names: { 1: n },  body:
    to: n,  apply:
      0: 1,  _:
        apply: *,  1: n,  2:
          apply: fac,  n:
            apply: decr,  1: n
apply: fac,  n: 3
```

Figure 5. Lisp, built around lists, vs. Masp, built around maps.

of source code files written in some text editor. In other words, it wisely takes advantage of the affordances of its Unix platform, avoiding the Turing Tarpit failure mode described in Section 7.2.

9.2 High-Level Language for BootstrapLab

If we take JavaScript, and strip away the concrete syntax, we get a resulting tree structure of function definitions, object literals, and imperative statements. A similar structure with similar semantics would be obtained from other dynamic languages. In fact, this would largely resemble Lisp S-expressions under Lisp semantics; hardly surprising considering Lisp's famously minimal syntax of expression trees. Furthermore, the evaluation procedure for Lisp is simple and well-established.

For these reasons, we designed a Lisp-like tree language in the substrate. This way, we provide high-level constructs (`if-else`, loops, functions, recursion, etc.) for in-system programming. Alignment (Force 3) encouraged us to revisit Lisp's design to better fit with our substrate. For example, ordinary Lisp is based on lists whose entries have *implicit* meanings based on their positions. This fits with the substrate made of S-expressions. Our substrate comes with named labels and suggests a language based around maps whose entries are explicitly *named*, so we called it *Masp*.[9] Figure 5 contrasts the two.

The equivalent Masp code is more verbose when rendered in ASCII. However, one of our key goals is to enable the use of other, better notations if desired, which we will discuss in the next section. Here, we start from an internal representation that has *more* information (explicit named arguments) than Lisp, but we can choose to display this however we feel appropriate (perhaps by showing a name label only for the entry being edited.)

[9]This is not too hard to come up with, but we would like to credit the origin of the name to [20] and related discussion.

9.3 Choosing an Appropriate Implementation

There are two basic decisions for implementing the high-level language. First, will we do it directly in-system using the instruction set, or using richer capabilities provided by the platform? Second, the language can be either interpreted or compiled. The four combinations have different properties.

A **platform interpreter** is the easiest one to implement, but it cannot be used to easily bootstrap itself. To "jettison" the platform implementation, we later need to *port* the interpreter to the ASM language. (Porting it to the high-level language would not suffice since we would still need the platform interpreter to actually run it.)

A **platform compiler**, while harder to implement, is slightly easier to jettison because it only needs to be ported to the newly developed high-level language. The platform compiler can translate it to ASM, which we can already run in-system. This compiler can then turn any high-level expressions into ASM, including a modified version of its source expressions!

Yet harder to implement is an **in-system interpreter**, directly in ASM, but it will exist in-system. The interpreter will, however, be less maintainable than if it were written in a high-level language and will likely need to be ported to a high-level language eventually.

Finally, an **in-system compiler** is the most challenging to implement. It will allow the language to exist in-system sooner and possibly more efficiently but, as above, will likely need to be converted to a high-level programming language to allow in-system improvements.

When implementing the interpreter or compiler in-system, all its intermediate state will also be stored in-system. However, in-system state can be used even when implementing the interpreter or compiler *on the platform.* This takes advantage of the platform's high-level language while leveraging the product system for debugging and visualisation, simplifying a later port to in-system implementation (see Heuristic 6). The transition from platform to in-system implementation can be even more gradual. Once the intermediate state is stored in-system, it becomes possible to port *parts* of the interpreter piecemeal to in-system instructions, invoking them from the remaining parts running outside.

Heuristic 6 (In-state operation)**.** Store high-level-language processing state in-system even if the processor runs on the platform. This will ease porting the processor to in-system implementation and support a gradual transition.

9.4 Implementing Masp for BootstrapLab

In BootstrapLab, Force 2 encouraged us to get executing Masp expressions early to get experience with the language.

We choose to implement a *platform interpreter* for Masp using JavaScript as this was the easiest way to achieve that.

A naïve implementation would simply implement the standard Lisp interpreter routines (`eval` and `apply`) as recursive JavaScript functions. However, this would miss an opportunity for visualisation and debugging that is already present in our substrate. Instead, we followed Heuristic 6 and had intermediate interpreter state reside in-system. This made a later in-system port easier by doing half of the work now.

Lisp evaluation is done by walking over the expression tree. At any point, we are looking at a subtree and will evaluate it until reaching a primitive value. Ordinarily, the "current subexpression" is an argument to `eval` at the top of the stack, while the stack records our path from the original top-level expression. Since we already had a tree visualisation, we used that instead of a stack. We did, however, need to maintain references to parent tree nodes (see Appendix B) in order to backtrack towards the next unevaluated subexpression once the current one is evaluated. Furthermore, instead of *destructively* replacing tree nodes with their "more-evaluated" versions, we "annotate" the tree instead. This design choice follows Subtext [6] and will make it possible to trace provenance and enable novel programming experiences.

What can be changed at the user level? Depending on which of the four implementation paths were chosen, the semantics of the language may or may not (yet) be modifiable from the user level. The user is almost able to use the high-level language in-system for convenient programming… but may be unable to enter the expressions conveniently in the first place. The latter will be addressed shortly.

10 Pay Off Outstanding Substrate Debt

> Port all remaining temporary infrastructure into the system, taking advantage of the infrastructure itself and the high-level language. The result is a *self-sustainable* programming system.

If we developed both the state editor and the high-level programming language in-system, we would already have an elementary self-sustainable system at this point. This would have been our only option if we had been somehow stuck with only a primitive platform, as was the case at the dawn of computing in the 1940s. With a richer platform available, one can choose to implement a state viewer, editor and high-level programming language on it following Heuristic 5. Since these will now run outside of the product system, they will be functionally part of the substrate, yet they do not belong there. This *substrate debt*, incurred due to Force 2, now needs to be paid off.

10.1 Substrate Debt in BootstrapLab

In the ideal development journey, we would have a high-level programming language and a basic state editor in-system by now. This did not happen for BootstrapLab.

The Masp interpreter we developed used in-system state, but controlled it from JavaScript. Our state viewer was also fully implemented in JavaScript. Editing took place through the browser development console. The alternative, creating a Masp interpreter and state editor in-system using the low-level ASM instructions, had been technically possible but prohibitively tedious. The in-system tooling was far from supplanting the existing platform interface of JavaScript in the text editor. Continuing to use the latter was, therefore, the only sensible choice to make progress.

Nevertheless, to make the high-level language and editor a part of self-sustainable programming system, they ultimately need to be implemented in-system. Thus we incurred a *substrate debt* due to Force 2 which we now need to pay off. The advantage of delaying this work is that we can at least port JavaScript to Masp, which is more convenient than using ASM. Generally, such substrate debt should be paid off as soon as the indebted implementation is complete. In total, we had three parts of it to pay off:

- The temporary state viewer, to be superseded by an in-system editor
- Its replacement state editor, to be ported from JavaScript to Masp
- The Masp interpreter, to be ported from JavaScript to ASM

In BootstrapLab, we took a two-step approach to supplanting the temporary state viewer. We first replaced a viewer that exists fully outside of the system with an editor that uses in-system state and graphics, but is controlled from JavaScript. We then started to port the editor code from the platform to in-system Masp, which is where we are at the time of writing.

10.2 Supplanting the Temporary State Viewer

Once we could run Masp programs in the substrate, we needed a better way of entering and editing them. We desired a state editor in the graphics window to make the existing state view obsolete. In-keeping with the proof-of-concept nature of this work, we created a rudimentary tree editor that nevertheless surpassed the existing practice of issuing commands in the JavaScript console.

To edit state in JavaScript, we needed to either address its parent with a full path from root, or to use a reference prevously obtained this way. To set a primitive value, we would type a JavaScript command including the key name and the value. This was not a high bar to clear. Evidently, we could greatly improve the experience by simply clicking on the relevant key name and typing.

We implemented a basic tree view in the graphics window (Figure 6). Nodes can be expanded and collapsed, and entries can be changed by clicking and typing. The display is "on-demand" and breadth-first: map entries are read upon expanding a node. This means that cycles in our graph substrate

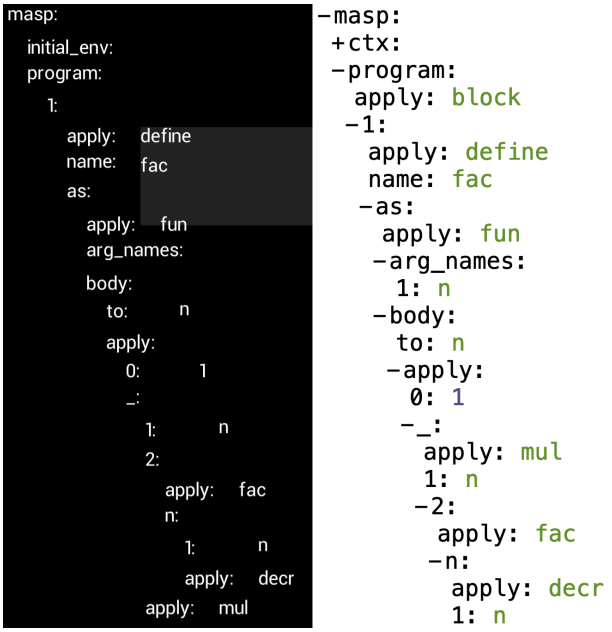

Figure 6. Left: tree editor in graphics window. Right: temporary state viewer in the DOM.

do not pose a fatal problem, as they did in the temporary state view (see Appendix B). The basic CRUD operations are accounted for as follows:

- Update (primitive): The `Tab` key commits the value and selects the next entry.
- Create: If the above runs past the end of the map, special "new entry" fields for entering a new key and value are created. These disappear if abandoned without committing.
- Update (composite): `Enter` commits a new, empty map and selects the "new entry" field within it.
- Delete: `Backspace` on an empty value will delete the entry. If it was the only entry, it will be replaced with the "new entry" field.
- Read: The display of the entry in the graphics window provides this.

It is worth noting that Alignment (Force 3) applies here too: the structure of the substrate clearly has implications for the structure of the editing interface. If our substrate consisted of low-level bytes, the traditional hex editor interface would be an immediate requirement. Such an interface could plausibly be simpler to implement than the complex nested tree editing we needed for BootstrapLab. This suggests a potential feedback into the choice of substrate: *a more complex substrate will require a more complex editor.*

We might even be tempted to conclude that it only makes sense to use a low-level substrate, since we can complete a basic editor sooner and subsequently work in-system. This neglects the fact, however, that the higher-level structures of our substrate would inevitably be required at some point. Thus we would have to do the same work anyway, but only once we had suffered through the human-unfriendly low-level substrate.

It is also remarkable that, in this restricted interaction domain, we finally *did* manage to surpass our default JavaScript text editor. There is a cost to typing out concrete syntax like `:` and `{}` for JavaScript map structures, as well as ensuring indentation is correct. For entering state structures, we found the structured editing style to be quicker. As a result, where previously we might have added new persistent state in our JavaScript startup code, we now directly entered it into the system and persisted it manually.

There is a caveat to all this. The whole exercise was in the service of paying our substrate debt from earlier—pulling up the state viewer "ladder" that had got us to this point. Ideally, we would have built up its replacement in-system. Yet as pointed out, JavaScript was still the most appealing way to program at this stage, so we used it for this editor as well. In other words, we took on a new debt in order to pay off the first one! To resolve it, we would port the JavaScript to Masp—a process which is underway at the time of writing for both the Masp interpreter and the state editor.

In general, at the end of this stage the substrate should not contain anything that we wanted to be modifiable in-system. Thus:

What can be changed at the user level? The structural "syntax" and semantics of the high-level language can be changed. The graphical interface of the system can also be changed, including the concrete notation for programs and data, which we turn to next.

11 Provide for Domain-Specific Notations

> Use the self-sustaining state editor to construct a more convenient interface for editing high-level expressions. Add *novel notations and interfaces* as needed. Use these not just for programming new end-user applications, but also to improve the product system itself.

Because BootstrapLab is currently in the middle of the previous stage, this section describes our plans for when this is complete. At such a point in the journey, the editor implementation is now part of the product system, so we can now modify it from within to our heart's content.

We admitted earlier how, in BootstrapLab, we had not managed to bring the system interface up to a level where it became more effective than JavaScript. With the implementation of a state editor, we came closer. Indeed, for entering general state structures, it is not obvious how to improve on it. Yet when it comes specifically to Masp expression structures, we must enter their verbose details even though they are highly regular and could be captured through fewer

interactions. If we streamline this *subdomain* of the BootstrapLab interface, it would make Masp programming just as convenient as typing JavaScript, if not more so—and we could finally escape the text editor entirely.

11.1 A Taster

First, we propose a restricted proof-of-concept of notational variation from within the system. We choose to target a small part of the problem: the Masp `apply` node, a frequent enough occurence that a small improvement will be helpful.

In the general state editor, one must type each of these key-value pairs for a function application:

```
apply: setColor
red: 11
green: 22
blue: 33
```

Instead, we desire something like autocomplete for parameters. Instead of typing `apply`, we press `a` and enter `setColor`. Subsequent tabbing should fill in the parameter names automatically and let us type the arguments. Furthermore, as a small notational difference we will omit the word `apply`:

```
setColor
red: _
green: _
blue: _
```

The underscores represent unfilled fields right after this structure gets created.

To reprogram the editor to work like this, we would do the following from within the editor. Navigate to the Masp code structures for the editor that synthesise the graphics structures to display a given state node. Enter Masp code that checks for the key `apply` in the given node and, if present, only renders the value of the key instead of the key itself.[10] Then, navigate to the code that handles key input. Add code that, when `a` is pressed, will insert a new map containing the `apply` key, render this to the graphics, and send text input to its value text box. Finally, navigate to the code that commits an entry on a `Tab` keypress. Change this to detect if it is for an `apply` key and, if so, to look up the symbol in the value node and treat it as a Masp function closure. For each entry in the `arg_names` field, add an entry to the map with a dummy value, render this to graphics, and then proceed with the default behaviour (highlight the next entry in the map). Depending on the precise implementation, it may be the case that only subsequent edits will be rendered this way. Otherwise, care may be necessary to refresh and re-render the entire editor state.

[10] Admittedly, this will display all structures with an `apply` key this way, but further discretion is just as achievable with further programming. The point is that this can be changed at the user level.

11.2 A More Ambitious Novel Interface

The above "taster" is a simple example of an interface that could be plausibly implemented early in BootstrapLab's self-sustaining lifetime. Beyond this, it points to a more general class of extensions which would support *projectional editing*. Projectional editors are a class of programming interfaces that provide domain-specific interfaces for certain program subexpressions, such as LaTeX-style mathematical expressions to replace ASCII renderings. We would do well to import such ideas into BootstrapLab. We proceed to sketch how such an interface would be added to the system, and how its ramifications are different from ordinary non-self-sustainable projectional editors.

As an example, suppose we want to program some fancy graphics. Fancy graphics require sophisticated vector mathematical formulae. In textual programming languages, these are expressed as ASCII with limited infix notation. The Gezira/Nile project [2, 3] attempted to improve on this with Unicode mathematical syntax. Obviously the extreme endpoint would be LaTeX. All we have at the moment is something worse than all of these: verbose, explicit tree views spanning multiple lines.

We think ahead with a view towards making the fancy graphics programming more pleasant. Suppose we decide that we would ideally like to implement them with the aid of concise mathematical notation, as opposed to our current state of verbose trees. How can we achieve this?

The broad approach would be similar to our previous taster example. We would have to start, again, at the code that renders state into graphics. Add a condition that checks for a `math` key, which we would use as a tag to hint at this display preference. Enter code to translate operator names to Unicode symbols, place them at infix positions, place parentheses appropriately, and render the whole thing to a single line in the tree view (ideally keeping the tree structure of the expressions in the graphics state). Then, modify the input handling and tree navigation code to appropriately work on this *inline* tree structure. And so on.

The above points are, of course, a high-level sketch, but it is *programming* all the same and is plausible to achieve with a high-level language. Techniques from the literature would be helpful, such as Hazel's calculus for editing structures with holes [17], or bi-directional synchronisation between the rendered graphics and the state's ground truth [10].

11.3 The Key Takeaway

In the non-self-sustainable world, a projectional editor is implemented in some traditional programming language and interface; say, Java. The domain-specific notations can benefit a wide variety of programs created using the editor. Yet, this range of beneficiaries nevertheless forms a "light cone" emanating out from the editor, never including the editor itself. For example, any vector formulae used to render the

interface of the editor will remain as verbose Java expressions, along with any code for new additions to the editor. The tragedy of non-self-sustainable programming is that it can never benefit from its own innovations.

Conversely, in BootstrapLab, the benefits of the new notation spread across the whole system; the "light cone" *includes* the editor implementation itself. If we previously had to squint and parse verbose maths trees in the implementation of the maths rendering, we can now open up the code again and see it rendered in the more readable way that it itself implements!

In COLA, notational variation appears to be limited to variation in concrete syntax. Our uncompromising insistence on *explicit, non-parsed structure* at the core of BootstrapLab, while costly in terms of interface implementation, was precisely in order to be free of such a restriction in the end. While one *could* implement a multiline text field with syntax highlighting in BootstrapLab, it is at least crystal-clear that a vast array of other interfaces are possible, unimpeded by any privileging of text strings.

12 Future Work

The final two steps of the journey described in the previous two sections constitute our own future work. There is, however, plenty of opportunity for follow-up work in the spirit of this essay and the project it has documented. At every step of the journey, there were choice points where we naturally could only move forward with one of the options. The obvious task for interested parties would be to explore the other branches. We can't provide an exhaustive listing here, but will give some important examples.

At the first step (Choose A Platform), all sorts of other platforms could be chosen. While COLA built on top of one "slice" of Unix—files, build tools and process memory—we see another possibility in focusing on the hierarchical *file system* as a state model to inherit through to a substrate. This is one obvious *structured* substrate lurking within Unix and some of our work here is no doubt applicable to it: directories act as maps, filenames as keys and file contents as values. Symlinks could add graph structure to this tree where needed.

We acknowledge that it might feel perverse to have files contain "primitive" values, such as a single number, or to represent instructions as directory trees, since files are normally used as "large" objects. However, it must be noted that there is precedent for using them more generally for data large and small, such as in Plan 9 and `procfs`. If this was still too much to stomach, the default option for "code", i.e. shell scripts, could simply be inherited (with the caveat that this would impose a text dependency at the core of the system). What is most unclear is how graphics would be displayed and interacted with—possibly requiring a special binary as part of the substrate, for opening and synchronising a main window.

If we accept our chosen web-based platform, we can consider alternative substrates. One obvious possibility is inheriting the DOM as the state model. This is the choice made by Webstrates [15], which stores textual JavaScript code for programmatic change. Following our approach, we might want a lower-level and structured instruction set instead. This would, at the very least, need to be capable of changing parent/child/sibling relationships, node attributes, and inner textual content. One warning is that the rest of the DOM API that would need to be exposed, in order to be able to produce a functional modern web page or web app, is somewhat daunting in scope. It would also be necessary to have some way of listening for changes to DOM nodes so that any constraints can be maintained or dependencies can be updated. Webstrates does provide synchronisation between networked clients on the same page, so perhaps its methods could be adapted.

As mentioned, we would also recommend going for an instruction set that is convenient enough to *use* that immediately building programs in-system is a worthwhile endeavour. Our own wild adventure in minimality was a mistake in this regard, causing us to stay in JavaScript, implement the high-level language there and port it later. It would be interesting to see the process of gradually building each component of a high-level language engine interactively in-system. Out of the four posibilities in Section 9.3, we chose the *platform interpreter*, so exploring the others would be illuminating—particularly the *platform compiler*, which could self-host relatively quickly.

Finally, it would certainly be interesting to forego any temporary infrastructure at all, or build up entirely in-system without using platform tools. This would require more careful substrate design to get this process going effectively. While this could give some insight or appreciation for the hardships of early computing, its practical value in the modern environment is unclear and may be best considered a challenge for hacker wizardry.

13 Conclusions

The process of developing a self-sustainable programming system that we followed in this essay roughly mirrors the historical development of programming that shaped much of how we do things today. Technology like the assembler and the compiler was born from a truly impoverished platform of flat memory, numerical instructions, printed output and rows of switches. Self-sustainable programming systems like Unix were gradually raised out of this primordial world, yet it still has a tendency to show through and force human minds to wrangle with it.

This essay can be seen as a sketch of how we might build similar infrastructure on the back of modern computing environments with structured representation of data and graphical interfaces. In other words, we investigate what

programming could look like if it were *re*-bootstrapped today, not on top of flat memory, but on a richer base platform such as that provided by web technologies.

In his 1997 OOPSLA keynote "The Computer Revolution Hasn't Happened Yet" [14], Alan Kay hoped that future users of Squeak/Smalltalk would use it to start a virtuous cycle of innovation: "Think of how you can obsolete the damn thing by using its own mechanisms for getting the next version of itself." It appears that these hopes were not answered in the 25 years since his keynote. Our modest contribution is to broaden the scope. If we weren't able to obsolete the damn *status quo* of text-only programming from Squeak, perhaps we can do it from any other platform—by following a handy sequence of steps.

A The Cutting Room Floor

Force 2 directed us to do without several advanced substrate features we were tempted to include. For example, it would be useful to attach state change listeners to keep parts of the state in sync with others. We could go even further and include constraint-based programming features.

On another note, our substrate is based on "maps" without a predefined ordering of the entries. However, there is always some order in which they will be displayed:

```
{ red: 100, green: 255, blue: 0 }
```

Thus it might be nice to be able to set this on a per-map basis. A convenient way to expose this in-system would be via another map, or "order map" which would be a list map of key names:

```
{ 1: 'red', 2: 'green', 3: 'blue' }
```

A practical use of this is for enabling iteration through a map's keys or entries. If we wish to be rigorous, the order map itself would have an order map, which would (by default) be the same for all order maps:

```
{ 1: 1, 2: 2, 3: 3 }
```

Of course, with such a conceptually infinite sequence of order maps, care must be taken to implement it in a finite, on-demand way. Perhaps some clever circular reference would work, as COLA does for its vtable relation [19]. This raises the question of how to obtain an order map in-system. If we make it an ordinary key on all maps, we must be careful to render it only on-demand and to exclude it from ordinary iteration through keys. Plus, would we want the visual clutter of always displaying it? It might be better to make it accessible through a special instruction `order-map`.

We then face a further *synchronisation* problem, where we must alter the display order whenever the order map is changed, and insert or remove entries from the order map to match its source.

Other thoughts along these lines included *parent* maps for delegating lookups (similar to JavaScript's prototype system), *inverse* maps, and *meta* maps for possibly collecting all of these (drawing inspiration from Lua's metatables). Of these, we will only discuss inverse maps in more detail.

Inverse maps come from the view of a map as a mathematical function from key names to values. Often in advanced data structures (such as those for graphical diagrams) it is essential to know "who points to me" via some key. For example, the question "Is this node the `source` of anyone else?" is a natural one, but normally it is impossible to answer based on ordinary dictionary keys. In ordinary programming languages, this information needs to be kept track of separately; say, in a manually synchronised list called `sources` that lives on the node. It is frustrating that the "forward" question is trivially answered by just following a map entry, yet the "backward" question has to be hacked around like this.[11]

An inverse map would somehow collect all references to a map from other ones. A user-level "map" would be implemented by two dictionary structures, the forward and backward halves, which are automatically kept in sync by the substrate implementation. The previously mentioned issues of access, mutation and others also rear their ugly heads here, so we can be forgiven for discarding the idea for the sake of making progress. Still, a properly worked out implementation would provide a valuable service for a high-level substrate.

B Graphs vs. Trees

A classic debate in the world of explicit structure is whether to use restricted *tree* structures or to allow arbitrary *graphs*. A tree has the advantage that every node has a single parent, which is a useful canonical answer to the question "what context am I in?". On the other hand, many practical problems do not fit inside a tree structure; either because they are DAGs, and a node can have multiple parents, or because they involve cyclic relations. Because we did not know what sort of things we would require in BootstrapLab, we erred on the side of freedom and supported full graph relations. This bit back at us in two ways, both involving the graphics domain.

Firstly, cyclic structures need to be rendered with care; a naïve depth-first search will never terminate. For a long time, we did not have any cyclic structures and got away with a depth-first approach to DOM generation in the temporary state view (Section 8.2).

Secondly, while this was the case, the graphics sub-region of state needed to be a tree. Spatial containment and other visual nesting (e.g. for the tree editor) is a tree structure, as is the underlying parent-child relationship of THREE.js objects. Many aspects of rendering the tree editor required the ability to ask "what context am I in?" but this is unanswered by default in a graph substrate. Providing a "parent" key for each node would not do—this would be a cyclic reference.

[11] Norvig's "Relation" pattern [16] for dynamic languages is relevant to this sort of concern.

Instead, we kludged it: the first map to reference another map becomes its "parent", and this lasts until the reference is deleted. This parent property is available from JavaScript; as we port the tree editor to Masp, we will have to decide how to expose it in-system (probably through a special instruction).

Of course, we eventually did require cyclic structures—for the tree editor! Each graphics node in the editor has a `source` key providing a way for edits to propagate back to the source state node. All edit nodes live in the graphics tree, including the one corresponding to the root node of the state. In this case, the `source` points all the way upwards to this root node. This cycle broke our state view and there was much gnashing of teeth to hack around this. Eventually, we bit the bullet and improved the state view JavaScript to cope with cycles—having previously hoped we were done with this temporary infrastructure.

Let this be a warning that Alignment (Force 3) will come for you in the end. If your substrate allows cycles, your state view must tolerate them!

C The Minimal Random-Access Instruction Set (And Its Perils)

Recall Heuristic 3 which instructed us to pursue an easy-to-implement instruction set. We pursued this goal to the extreme out of curiosity for what was possible. Of course, it turned out that the corresponding explosion in the number of instructions necessary to do a simple thing outweighed any implementation advantage…

We did this by breaking down higher-level instructions to their component operations until we felt we could go no further. This led to a sort of "microcode" level where each instruction's implementation corresponded to some single-line JavaScript operation. In other words, the platform itself blocked any further decomposition.

Our method for achieving this can be illustrated if we start with a hypothetical complex instruction, e.g. `copy a.b.c to x.y.z`. The actual *work* involved in executing this in JavaScript would involve three steps:

1. Traverse the path `a`, `b`, `c` and save the value in a local variable
2. Traverse the path `x`, `y` and save the (map) value too
3. Set the key `z` in the map to the saved value.

If we score *strictly by JavaScript implementation size* (a mistake, in hindsight), we could improve by simply splitting up these steps into instructions of their own. Any other "complex" instructions that used some of the same steps (e.g. path traversal) will also be covered by these, and the total JavaScript will be reduced.

For the first path traversal, we start at the root map (or more generally, any given starting map) and follow each of the keys in turn. We have only one step here (follow key) repeated three times. That's another micro-instruction!

At this stage, we have this truly simple instruction: `follow-key k`. It clearly relies on some implicit state register for the current map, and takes a single parameter. We pushed the limits of sanity by going further, factoring the parameter *out* into another state register, so the resulting instruction is just `follow-key` (we called it `index`). In other words, we applied the following heuristic:

Heuristic 7 (Registers for parameters)**.** Factor out instruction "parameters" into special state registers where possible.

The motivation for this is a vague intuition about sharing parameter values. Under a parameter scheme, copying the same thing to multiple destinations will duplicate the "source" parameter many times, even though the only thing that's changing is the destination. The converse is true for operations with the same destination—maybe not overwriting copies, but arithmetic or other accumulating operations. By breaking these parameters into state, we set a source or destination once only. This has a subjective aesthetic appeal from the point of view of minimality, and an even more dubious efficiency value. We emphasise that it was an experiment and advise against it for the purposes of implementing a system quickly.

As mentioned at the end of Section 7.3.1, we end up with only four registers (`next_instruction`, `focus`, `map`, `source`) and five[12] core instructions (`load`, `store`, `deref`, `index`, `js`). These have a structural representation in-system, but also a convenient textual syntax for brevity in textual media (like this essay).

Combinations of these express the expected copying and jumping operations. For example, `load source-reg`, `deref`, `store dest-reg` copies the value in top-level `source-reg` to `dest-reg`. The first instruction loads the literal string `source-reg` into the `focus`; the second replaces `focus` with the contents of its named register; the third copies the `focus` to the named destination.[13]

The `copy a.b.c to x.y.z` from earlier would decompose as follows:

1. `load a`, `deref`, `store map`, `load b`, `index`, `load c`, `index`, `load map`, `deref`, `store source`
2. `load x`, `deref`, `store map`, `load y`, `index`
3. `load z`, `store`

(Recall that `index` replaces `map` with the result of following its key named by `focus`, and `store` without any arguments copies from `source` to the `focus` key entry within `map`.)

[12] Or six, if we analyse the overloaded `store` instruction as `store-reg` and `store-map`.

[13] It turns out that, if you extract the destination parameter from `store`, you meet an infinite regress and will be unable to store to any top-level register. For example, if we extract the parameter to `dest_reg`, we have to somehow give it the value it previously took in the instruction—but this is precisely a `store` operation and we're already in the middle of one.

A jump is accomplished by overwriting the address in `next_instruction` (i.e. a map containing a `map` field and a `key` field). The map or the key can be overwritten in a single instruction, but if an entirely new address is required, this needs to be built up separately and overwritten atomically. In other words, we cannot overwrite the `map` and then overwrite the `key`. The ugly reality is, after overwriting the `map`, it will have jumped to a different instruction somewhere else!

A conditional jump is sneaked in by indexing a map to obtain the new list of instructions (which is the map that will overwrite the `map` under `next_instruction`). For example, in the following register snapshot:

```
...
weather: 'stormy'
map: {
    sunny: { ... sunny code sequence ... }
    rainy: { ... rainy code sequence ... }
    _:     { ... other code sequence ... }
}
```

One of the three code paths will be selected according to whatever happens to be in the `weather` register via the following instructions: `load weather`, `deref`, `store focus`, `index`. The `map` register will hold the result, in this case the "other" code sequence (recall that the special key `_` is used as an "else" clause for lookups). What remains is then to copy this within `next_instruction`.

It is easy to see how this applies for strict equality matching, but what about comparisons? We simply turn the condition into one of a fixed set of constants. For $3 < 7$ we would subtract to get -4 and then apply the mathematical `sign` function to obtain -1 (the other possibilities being 0 or 1). we would then index a map containing keys `-1`, `0` and `1`.

Finally, operations like subtraction and `sign` were included as special instructions or achieved via the `js` escape hatch into JavaScript. We continued to experiment with other arithmetic instructions, including vector arithmetic (useful for graphics), but never got round to implementing an operand stack.

References

[1] Kartik Agaram. 2020. Bicycles for the Mind Have to Be See-Through. In *Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming* (Porto, Portugal) *(<Programming> '20)*. Association for Computing Machinery, New York, NY, USA, 173–186. https://doi.org/10.1145/3397537.3397547

[2] Dan Amelang. 2014. *Gezira*. https://github.com/damelang/gezira

[3] Dan Amelang. 2014. *The Nile Programming Language*. https://github.com/damelang/nile

[4] Marcel Borowski, Luke Murray, Rolf Bagge, Janus Bager Kristensen, Arvind Satyanarayan, and Clemens Nylandsted Klokmose. 2022. Varv: Reprogrammable Interactive Software as a Declarative Data Structure. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems* (New Orleans, LA, USA) *(CHI '22)*. Association for Computing Machinery, New York, NY, USA, Article 492, 20 pages. https://doi.org/10.1145/3491102.3502064

[5] Tim Colegrove. 2020. Own work, CC BY-SA 4.0. https://commons.wikimedia.org/w/index.php?curid=89430810

[6] Jonathan Edwards. 2005. Subtext: Uncovering the Simplicity of Programming. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (San Diego, CA, USA) *(OOPSLA '05)*. Association for Computing Machinery, New York, NY, USA, 505–518. https://doi.org/10.1145/1094811.1094851

[7] Richard P. Gabriel. 2012. The Structure of a Programming Language Revolution. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Tucson, Arizona, USA) *(Onward! 2012)*. Association for Computing Machinery, New York, NY, USA, 195–214. https://doi.org/10.1145/2384592.2384611

[8] James Hague. 2010. *Living Inside Your Own Black Box*. https://prog21.dadgum.com/66.html

[9] Christopher Hall, Trevor Standley, and Tobias Hollerer. 2017. Infra: Structure All the Way down: Structured Data as a Visual Programming Language. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Vancouver, BC, Canada) *(Onward! 2017)*. Association for Computing Machinery, New York, NY, USA, 180–197. https://doi.org/10.1145/3133850.3133852

[10] Brian Hempel, Justin Lubin, and Ravi Chugh. 2019. Sketch-n-Sketch: Output-Directed Programming for SVG. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology* (New Orleans, LA, USA) *(UIST '19)*. Association for Computing Machinery, New York, NY, USA, 281–292. https://doi.org/10.1145/3332165.3347925

[11] Robert Hirschfeld, Hidehiko Masuhara, and Kim Rose (Eds.). 2010. *S3 '10: Workshop on Self-Sustaining Systems* (Tokyo, Japan). Association for Computing Machinery, New York, NY, USA.

[12] Robert Hirschfeld and Kim Rose (Eds.). 2008. *S3 '08: Workshop on Self-Sustaining Systems* (Potsdam, Germany). Springer. https://doi.org/10.1007/978-3-540-89275-5

[13] Joel Jakubovic, Tomas Petricek, and Jonathan Edwards. 2022. Technical Dimensions of Programming Systems (forthcoming). (2022).

[14] Alan C. Kay. 2000. The Computer Revolution Hasn't Happened yet (Keynote Session). In *Proceedings of the Eighth ACM International Conference on Multimedia* (Marina del Rey, California, USA) *(MULTIMEDIA '00)*. Association for Computing Machinery, New York, NY, USA, 1. https://doi.org/10.1145/354384.354390

[15] Clemens N. Klokmose, James R. Eagan, Siemen Baader, Wendy Mackay, and Michel Beaudouin-Lafon. 2015. Webstrates: Shareable Dynamic Media. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology* (Charlotte, NC, USA) *(UIST '15)*. Association for Computing Machinery, New York, NY, USA, 280–290. https://doi.org/10.1145/2807442.2807446

[16] Peter Norvig. 1996. Design patterns in dynamic programming. (1996). https://norvig.com/design-patterns/ Object World, Boston, MA.

[17] Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. 2019. Live Functional Programming with Typed Holes. *Proc. ACM Program. Lang.* 3, POPL, Article 14 (jan 2019), 32 pages. https://doi.org/10.1145/3290327

[18] Ian Piumarta. 2006. *Accessible Language-Based Environments of Recursive Theories*. http://www.vpri.org/pdf/rn2006001a_colaswp.pdf

[19] Ian Piumarta and Alessandro Warth. 2006. *Open, Reusable Object Models*. http://www.vpri.org/pdf/tr2006003a_objmod.pdf

[20] C2 Wiki. 2014. *Masp Brainstorming*. https://wiki.c2.com/?MaspBrainstorming