CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

JS	JavaScript

A BOOTSTRAPLAB SUBSTRATE REFERENCE

This chapter serves as a more complete description of the BootstrapLab substrate than the one given in Section **??**. Following Section **??**, we divide it into a discussion of *State* and *Change*.

A.1 STATE IN BOOTSTRAPLAB

State is a graph of *maps* (key-value dictionaries) pointing to each other, along with JS *primitives:* numbers, booleans, strings, `null`, `undefined`, and ordinary JS objects. A *list* is a map with numerical keys. Like in JS, this does not prevent it from having non-numerical keys at the same time.

Maps contain *entries* or *fields* consisting of a *key* and a *value*. We also refer to maps and primitives together as *values*. Starting from a given map `m`, we write a *path* `m.foo.bar.baz` to denote the value arrived at by following the keys `foo`, `bar`, and `baz`. For this to be defined, `m.foo` and `m.foo.bar` must be maps.

The substrate inherits the reference semantics of JS; if one wants to insulate a map from side effects originating from other references, one must perform a copy to some level of depth and operate on the copy.

There is a *root* for the state, from which everything is discoverable; all absolute paths begin here, and every value that "exists" necessarily has some path from root, although it may have more than one owing to the graph structure. An "address" consists of a map and a key; analogously to paths, a piece of state may have more than one "address" if it is referenced from multiple maps.

Because values at the root level are accessible from only a single key, they are the "first port of call" for instructions and are known as *registers*. Some specific *substrate* registers are reserved for use by instructions. The main substrate registers are:

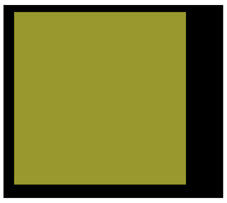
- `focus`: an "accumulator" register used by most instructions.
- `map`: holds the map to be updated or navigated by map instructions.
- `source`: holds the new value when updating a map entry.
- `next_instruction`: program counter; holds the address of the next instruction and the instruction itself.
- `scene`: container for the scene tree affecting the graphics window (discussed shortly).

All other keys at the root level are general-purpose *user registers* available for programs to use.

A.1.1 *Graphics State: The `scene` Tree*

While state in general can be graph-structured, it is tree-structured[1] under the `scene` register. The `scene` itself is a list of *graphics maps*. The substrate recognises a map as a graphics map if it contains at least one *graphics property:* any of the keys `width`, `height`, `color`[2], `opacity`, `position`, `center`, `top_left`, `zoom`, `text`, or `children`. The presence of these keys causes the substrate to connect their values to shapes in the graphics window and maintain synchronisation. Any graphics map can have a list of `children` in the scene hierarchy.

At present, only crude graphics are possible via two shapes. A rectangle has a `center`, `color`, `width`, and `height`, while a text label has a `top_left`, `text`, and `opacity`. For example:



```
-yellow_shape:
  color: 0x999900
  width: 2
  height: 2
 -center:
   basis: shapes
   right: 2
   up: 1.750
   forward: -1
```

```
-mytb:
  text: Hello World
 -top_left:
   right: 1.500
   up: 0.5000
```

As can be seen, `color` accepts a hex string; `opacity`, not shown, is a number between 0 and 1. More interesting is the protocol of vector properties like `center` and `top_left`. Firstly, the naming of these properties themselves follows a design principle which could be called "Naïve Honesty".[3]

Heuristic 1 (Naïve Honesty)**.** If a more abstract term only has one intended meaning in a particular API, name based on the honest concrete concept instead.

For example, the term "position", widely used in graphics APIs, is not very self-documenting when it comes to shapes. In practice, it always means "the centre of the shape" or "the top left-hand corner" or some such meaningful function of the shape itself. Therefore, why not just be explicit about this, and save the user the effort of figuring out what is actually meant? The co-ordinates are specified according to the same principle: instead of the nebulous `x` and `y`, we have `right` and `up`. In an ideal substrate, one could specify co-ordinates via `left` and `down` or any combination, and the system would automatically flip the signs as necessary for its platform graphics API calls. However, this functionality does not yet exist in BootstrapLab.

1 See Section **??** for how this was accomplished.
2 We comply with the fact that, for better or worse, American spelling conventions are the *de facto* standard for internal identifiers and code more generally.
3 By analogy to Boxer's "Naïve Realism".

Third, a vector property may have a `basis` key naming a registered co-ordinate frame. This can be seen in the special *camera* graphics map, linked to the zoomable/pannable view in the graphics window:

```
-camera:
  zoom: 0.4403
 -position:
   basis: world
   right: 1.451
   up: -0.5981
   forward: 10
```

If unspecified, the basis is assumed to be that of the parent node in the tree. However, the user can set an explicit `basis` to express co-ordinates as most convenient to them, while the substrate will convert between frames under the hood. This is another application of Naïve Honesty in regard to more of our frustrations about graphics programming: the co-ordinate basis of a vector is often left as *implicit* information for the programmer to carry around in his head, who must also have a notebook handy to write the relevant matrix equations to transform things properly. We believe this tedium is exactly what should be handled automatically by the substrate (on this theme, see Geisler et al. (2020)). As it stands, bases are registered by the key name of the graphics map into a flat list, and are thus vulnerable to name collisions and synchronisation issues. However, what we have is a promising start.

Naïve Honesty can be seen as a response to a design requirement called "No Guessing":

Requirement 1 (No Guessing)**.** The user of an API should never have to uncover the meaning of a concept through trial-and-error experimentation. In graphics programming, the user should never have to create a throwaway object and transform it to resolve ambiguity about sign conventions, axis conventions, unit conventions, basis conventions, etc.

A reasonable reply to No Guessing might be to just write better documentation. However, this is not a solution to implicit bases in user code, and Naïve Honesty attacks the root of the problem in the poor conventions themselves.

A.2 CHANGE IN BOOTSTRAPLAB

In-keeping with Alignment (Force **??**), the smallest units of change ought to just "fall out" of the structure of the State:

- Change a map entry to a new value (our `store` instruction)
- Create a new map (our `load` instruction)
- Actions necessary to support the above (`index` and `deref`)

- Inheritance of platform changes, i. e. mutate JS state and call any API (`js`)

We will elaborate on these shortly, but first we will specify how instructions are represented as state, and how we will notate them in shorthand and diagrams. We will mostly stick to reference material here; for more in-depth design rationale, see Section A.3.

A.2.1 *Instruction Encoding In State, Text, and Diagrams*

An instruction is a map with an `op` field serving as the opcode, along with any parameters as further entries. Some examples:

```
{ op: load, value: my_reg },
{ op: deref },
{ op: store, register: my_dest }
```

Because this notation is so verbose, and no instruction has more than one parameter (see Section A.3), we use an inline textual notation with unnamed parameters. We would write the above like so:

```
load my_reg ; deref ; store my_dest
```

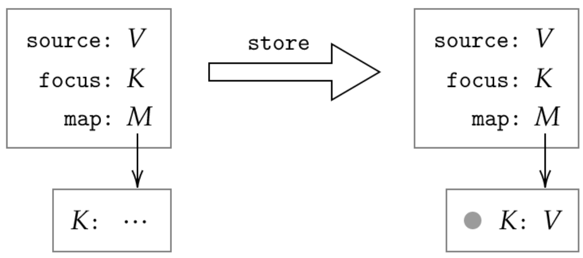
In the next section, we will specify the semantics of the instructions. In line with the spirit of *Notational Freedom* (Section **??**), we will use box-and-arrow diagrams, since we judge this more suitable for our substrate than traditional formal semantics notations. However, we will take some cues from the latter; for example, we show the state before and after the instruction. We also use symbols to stand for abstract values. Specifically, K denotes any string (or "key"), M denotes a map, and V denotes an arbitrary value. We additionally employ a grey spot to highlight the part of the state that was mutated.

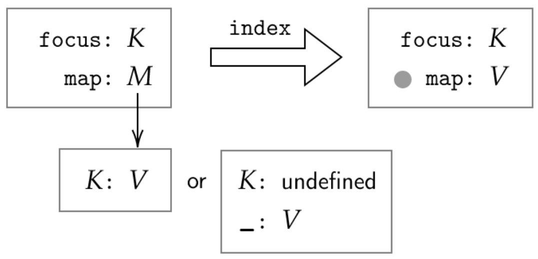
A.2.2 *Change Map Entry and Supporting Instructions*

The `store` instruction, with no parameters, expects a map M in the `map` register, a key string K in the `focus` register, and a value V in the `source` register. After execution, the entry K of M will have the value V.

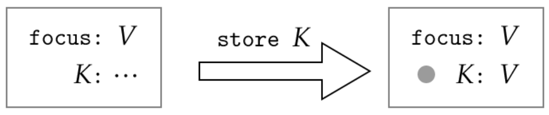
The `index` instruction, like the `store` to which it is dual[4], takes a map M in `map` and a key string K in `focus`. After execution, `map` contains the value $M.K$, unless this is `undefined`. In that case, it will try the special key _ as failsafe and `map` will contain $M.\_$ instead, which could still be `undefined`.

4 We mean this in an informal sense, but it points to some interesting analysis which we have not undertaken. Compare also `deref` and the register version of `store`, leaving `focus` curiously on its own.

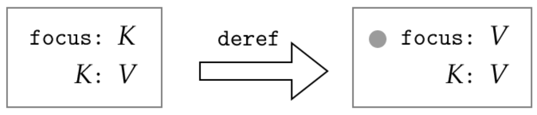




There are alternate semantics[5] when `store` is executed with a parameter `register` of string value K. In this case, given a value V in `focus`, the root-level entry K will have value V.



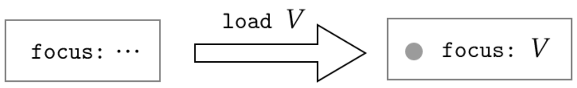
With a string K in `focus` and register K holding the value V, an execution of `deref` will place V in `focus`.



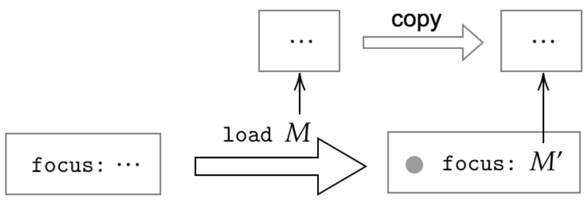
Finally, `load` takes a parameter `value` with a value V. After execution, the `focus` register contains V.

Actually, there is a subtlety if V is a map: the `focus` register then contains a *copy* of V. This is to preserve the intended use of the `value`

5 This "overloading" of an instruction is straightforward in a map substrate, as compared to a flat binary one (Section **??**), although an argument could be made for it to be a separate instruction called `store-reg`.



parameter as a "literal"[6], especially when the map is empty (see Section A.2.3 shortly).



At the time of writing, the copy performed is a deep copy, but this is almost certainly wrong, as would be always using a shallow copy. The purpose of having a `load` instruction with a map literal in the code is to *instantiate* a new map with that structure, so the literal in the code must remain unaffected. The basic intuition (Figure A.1) is that we want things that *present as* part of the literal in the code to be fully (deep) copied; in other words, an inner map that is fully "nested" (and not referred to elsewhere) should be duplicated. Meanwhile, any references to data structures that already exist elsewhere (which thus should not present as "fully nested" in the map literal) should be preserved. However, the substrate does not yet distinguish between these two cases.[7]

A.2.3 *Create New Map*

Because of the fact that `load` makes a copy of its `value`, creating a new map is simple: `load` M, where M is an empty map. After execution, `focus` contains a new empty map that can be written to, *without* affecting the empty map in the instruction itself, which will stay empty (unless, of course, the instruction is deliberately modified "as data" by separate code). If `load` did not perform a copy, it would be necessary to use a *different* `load` instruction each time one wanted to create a new map.[8] However, arguments could be made to the effect that `load` should just

6 We use "literal" by analogy to "string literal", "number literal", etc. In other words, an entity presented in its entirety in the source code, rather than loaded from an external source at run time or built up from separate pieces.

7 It could be argued that the generator of the code (whether user or compiler) knows best, hence `load` should just take an optional flag for whichever case is not the default.

8 This situation is the same Python's notorious "mutable default arguments" for functions (Python Guide 2016). Default arguments are evaluated the one time a function gets *defined*, instead of per call, so subsequent calls to the function will see any successive changes to the same default argument object.

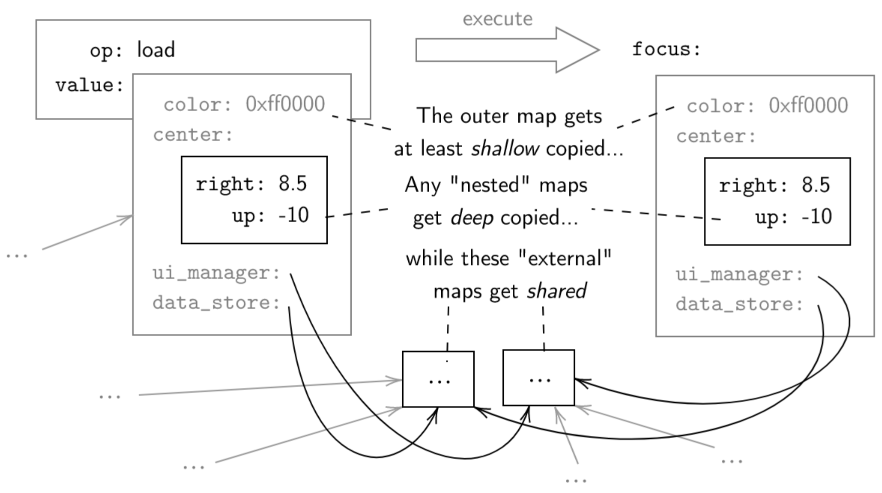

op: load
value:
color: 0xff0000
center:
right: 8.5
up: -10
ui_manager:
data_store:
execute
focus:
The outer map gets at least *shallow* copied...
Any "nested" maps get *deep* copied...
while these "external" maps get *shared*
...

Figure A.1: On the left, we have a `load` instruction for a ”literal” map, meant to be used as-is. Its effect is to fill the `focus` register with *some sort of* copy of the `value`. We would present “child” data (i. e. with no other parents/referrers) as graphically nested within its parent, and these maps should get duplicated. However, data that is clearly meant to be shared, as evidenced by other existing references, ought to remain that way.

have special semantics for the empty map, or that there should be a `load-new` instruction, and so on.

A.2.4 *Inheritance of JS-level Change*

The `js` instruction takes a `func` parameter and calls it as a JS function. This functions as an all-purpose “escape hatch” into the JS platform, analogous to `asm` blocks in C code.

```
{ op: js, func: () => {
    alert("Hello, World!");
}}
```

A.2.5 *The Fetch-Execute Cycle*

The `next_instruction` register holds a map with keys `ref` and `value`. The `value` entry hold the next instruction itself, while `ref` contains entries `map` and `key` as the address of the instruction, where `key` is an integer. During the fetch-execute cycle, `key` is incremented. In this way, a list of instructions can serve as a “basic block”. Furthermore, if incrementing the `key` “runs off the end” of a list, the substrate will follow any `continue_to` entry in the list and continue execution at key `1` of the associated map.

A.3 THE MINIMAL RANDOM-ACCESS INSTRUCTION SET (AND ITS PERILS)

In this section, we will explain the design rationale that led us to the peculiarities of the instruction set. Recall Heuristic **??** 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 JS 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 JS 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 JS 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 JS 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 very 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 2 (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.

A.3.1 *Copying And Jumping*

Combinations of the instructions express the expected copying and jumping operations. For example, `load source-reg,deref,store dest-reg` copies the value in root-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.[9]

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`.)

A jump is accomplished by overwriting the address in `next_instruction.ref` (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* into a map to obtain the new list of instructions (which is the map that will overwrite the `map` under `next_instruction.ref`). For example, in the following register snapshot:

```
...
weather: 'stormy'
map: {
```

9 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.

```
    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.ref`.

This example proved that we can do strict equality matching, but what about comparisons? We may observe how conditional jumps are traditionally based on a relation to zero, e. g. "jump if less than zero". A condition like $3 < 7$ is rearranged into $3 - 7 < 0$ and the "jump if less than zero" instruction is called with the value -4. In BootstrapLab, we go one step further: if we apply the mathematical `sign` function to the value -4, we get -1. We can then simply use the trick we already described and `index` into a map containing keys `-1`, `0` and `1`. All of the relevant conditions can be expressed this way: for example, "greater than or equal to zero" can be achieved by pointing the keys `0` and `1` to the same code sequence.

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

A.3.2 *Proof of Turing-Completeness*

because i say so

A.3.3 *Remarks*

Fails the self-interpret test

See also blog post