Ringfuck: It's a wheel great time!

Ryan Hornby and Matthew Safar

No Institute Given

Abstract. Left as an exercise to the reader.

1 Introduction

The programming language Brainfuck has been helping programmers improve their skills since its creation in 1993 by Urban Müller. Its great design of simplicity and power have left it unchanged since its inception, unlike many modern "high level" languages. Recently, Sigbovik contributor Ho Thanh has expanded the possibilities of Brainfuck with the improved functionality version Brainfuck++, which includes object oriented programming (it doesn't actually), networking, and many quality of life features (Thanh, 2022). Unlike these attempts to modernize Brainfuck, however, we propose a radically new innovation to the Brainfuck language that addresses some gaping (circular) holes in the language.

Brainfuck programs operate on a tape of memory, similar to a Turing machine. However, as anyone who has gotten super high and watched *Arrival* will tell you, the true form of everything is a circle (Garland, 2015). Whether it be physical circles, such as spherical cows (Stellman, 1973) or temporal cycles such as the boom and bust cycle of our great capitalist economy, the most natural shape is the one that doesn't cut any corners: a circle. To this end, we have developed a new offshoot of Brainfuck, which we call Ringfuck which allows you to practice your circular reasoning.

2 The Ringfuck Language

2.1 Brainfuck Syntax

Fundamentally, this language is based on the syntax of Brainfuck. As previously described, the Brainfuck language is comprised of only eight commands, listed in Table 1. Any Brainfuck program is simply a list of these eight commands executed sequentially, which together move a data pointer that modifies and reads from a tape of memory, segmented into bytes.

It's this simplicity that has prompted prominent experts in the field of computer science to give Brainfuck the venerable title of a "Turing tarpit" (Perlis, 1982).

`+`	Increment the byte at the data pointer.
`-`	Decrement the byte at the data pointer.
`>`	Increment the data pointer.
`<`	Decrement the data pointer.
`[`	If the value at the data pointer is zero, jump to the command after the corresponding `]` command.
`]`	Jump back to the corresponding `[` command.
`.`	Output the byte at the data pointer.
`,`	Access one byte of input and store it at the data pointer.

Table 1. The eight commands in Brainfuck.

2.2 Rings

Instead of using tape, we propose that memory should be folded onto itself, and connected into rings, similar to how you would fold tape to stick a polaroid of that one time you went to Europe in college above your bed.

Header Realistically, any Brainfuck program is actually just at RingLang program on a single infinite ring. However, this one ring to rule them all approach is boring, and, in practice, impossible, so we include a header to specify ring sizes. Note that Ringfuck supports (and encourages! `<3`) having multiple rings.

This is done by writing the desired size of each ring separated by white space at the top of the file before any instance of the eight commands.

Brainfuck Commands with Rings For a single ring, the Brainfuck commands are the same, with the exception of `>` and `<`, which now loop around the ring.

Multiple rings, however, necessitates multiple data pointers. Brainfuck commands affect all data pointers, meaning that each ring will execute the whole program individually. The input and output commands (`,` and `.`) read and write according to the order of initialization. Notably, the input command requires as many inputs as there are rings, and will wait until each of the selected bytes are filled.

A Simple Example We'll start by looking at the canonical example of a program in a new language, the "Hello World" program. The below code will print "HELLO WORLD! \n" when executed. This is done by loading index 0 cells of each ring with the ascii codes for the appropriate character and then printing them out. Note that the below program is slightly optimized and might be slightly unintuitive, if this is the case please see the Appendix for an unoptimized version of this program.

```
3 4 5
<++++++++[>+++++++++<-]>>>>>--->++++<<<<<.++++>>>>++++++>-<<<<<<<
<<<<<<<<<+++[>--------<-]>+>>>>>>>>>>.+++++++++++>>>>>>>>--------
---<<<<<<<<<<<<<<<<<<<+++++[>++++++++<-]>->>>>>>>>>>.-----------<
<<<<<<<<<<+++++[>--------<-]>++>>>>>>>>>>.-----------------------
--------------------->>>>++++++++>+++++++++++++++++++++<<<<<.
```

Now that we all know how to write a "Hello World" program in Ringfuck we can all put it on our resumes and call ourselves experts.

The astute among you may be wondering about mutually prime wheel sizes or the possible state spaces of a set of wheels, if so then this brief mathematical interlude is just for you!

A Brief Mathematical Interlude A ringset is a set $S = \{R_0, \ldots, R_{m-1}\}$, where each of the rings R_i has n_i elements $\{x^i_j \in D; j = 0, \ldots, n_i - 1\}$ (we will use superscripts on values to denote which ring they come from, and subscripts to denote their place on the ring). The domain D is, in our case, the set $\{0, 1, \ldots, \mathrm{ff}_{\mathrm{hex}}\}$ for computer reasons.

All of the operations in a program affect the "indicated" value, which takes one value from each ring in S. Therefore, if the indicated index is k, then the current value of ring R_i is $x^i_{k \pmod{n_i}}$, and the indicated value of the ringset as a whole is the ordered sequence $X = (x^0_{k \pmod{n_1}}, \ldots, x^{m-1}_{k \pmod{n_{m-1}}})$.

In the language described above:

– `>`/`<` increment and decrement the indicated index k,
– `+`/`-` increment and decrement the values $\{x^i_k; i \in 0, \ldots m-1\}$ all together,
– `.` prints each of $(x^0_{k \pmod{n_1}}, \ldots, x^{m-1}_{k \pmod{n_{m-1}}})$ in `ascii` encoding ordered as they appear in the sequence,
– and `,` takes external values $(y^0, \ldots, y^{m-1})$ and inserts them into the indicated values X.

A helpful way of expressing the positions of the rings is through their individual indicated indices, the ordered sequence: $I = (k \pmod{n_1}, \ldots, k \pmod{n_{m-1}})$. Due to the coupling of the rings under `>`/`<`, the ringset as a whole can be thought of as a ring of memory, with the amount of increments to k required to return to the same place being the least common multiple K of the elements of I. We call a ringset *pleasant* if the size $K = \prod n_i$, i.e. if all the ring sizes are mutually prime.

For later, it will be convenient to remember that ringset positions are only possible to achieve if for all $i, j \in \{0, \ldots, m-1\}$ they satisfy

$$I_i \equiv I_j \pmod{\gcd(n_i, n_j)}.$$

From here, we arrive on the most interesting aspect of the ringset, which is the shared memory between different X's. We say that two ringset indices k_1 and k_2 are i-linked (denoted with $\overset{i}{\leftrightarrow}$) if the ith ring has the same indicated index. In other words:

$$k_1 \overset{i}{\leftrightarrow} k_2 \iff k_1 \pmod{n_i} = k_2 \pmod{n_i}.$$

We say two indices are $\{i_1, i_2\}$-linked if they are both i_1-linked and i_2-linked.

We say that two indices are linked if there exists an $i \in \{0, \ldots, m-1\}$ such that they are i-linked. Otherwise they are not linked.

To gain intuition about memory linkage, we provide graphs, see Fig. 1. The figure represents different linking by different colors. The graph of the color representing the ith ring is made up of n_i separate complete graphs of size K/n_i, offset by one each time. (Naturally, one can easily consider the notion of ij-linking which considers the linking of the jth position of the ith ring, which is just a subset of the larger i-link equivalence relation). Since the memory linkage is predetermined from the size of the rings, constructing a useful memory structure is somewhat finicky, but possible.

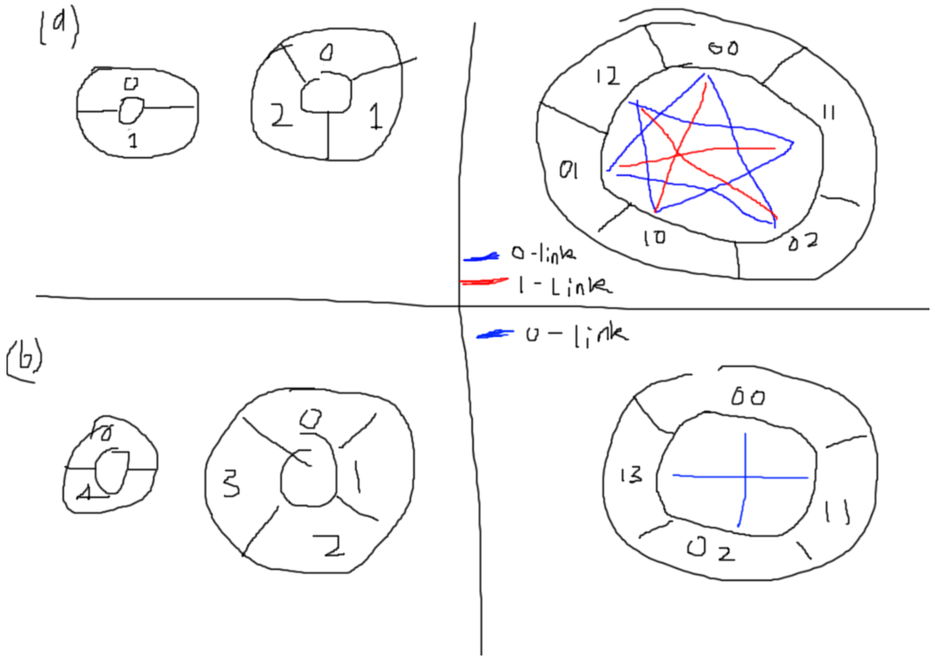

Fig. 1. (a) rings, combined rings with links for 2 3. (b) rings, combined rings with links for 2 4.

Note that for non-pleasant ringsets, the memory-linkage graphs are constrained by the fact that many ringset positions are not possible. An important result of this restriction is the result of several separate complete graphs within the total linkage graph. Fig. 1 (a), for example, is pleasant and has a single complete linkage graph, when all edges are combined, while in (b), there are two complete but not connected graphs formed by the vertices 00 and 02, and 11 and 13. This separation is easily described by the equivalence relation where two vertices $I_i = (x_i^0, \dots x_i^{m-1})$ and $I_j = (x_j^0, \dots x_j^{m-1})$ are in the same complete graph if and only if

$$x_i^s(\pmod{\gcd(n_s, n_t)}) = x_i^t(\pmod{\gcd(n_s, n_t)}), \qquad \forall s, t \in \{0, \dots, m-1\}.$$

For reasons evident later, we call the vertices of these separated graphs *offset families*

Another important aspect of the ringset structure relates to how the `+`/`-` operators affect m values simultaneously. This causes some complications as to which ringset configurations are possible.

For example, let's consider for now rings with the domain $D = \mathbb{Z}$, so that there is no wrap-around effect due to overflow. For a single ring, any possible configuration of numbers is possible using the `+` and `-` operators.

Also note, there is a minimal number of `+` and `-` operators needed to reach a given configuration from the configuration of all zeros, but no maximum number of operators, as they cancel each other out. This means that a program can achieve a given configuration from the zero configuration only if the number of `+` operators N_+ and `-` operators N_- is such that $N_+ - N_- = \sum x_i$, the sum of all the values in the configuration. This will be important later.

From here, consider the case of a pleasant ringset with two wheels. Now, it is no longer possible to achieve any possible configuration of the ringset because of the coupling between the two. Since the operators act on both wheels, the offset between the sums of their configurations is invariant:

$$\sum x_i^0 - \sum x_i^1 = \text{const.}$$

However, we will show that this invariant (which we will refer to as the *offset invariant*) is a necessary and sufficient condition for a particular configuration to be reachable from a given starting configuration.

($\Rightarrow$) **Any given program preserves the offset.** See above.

($\Leftarrow$) **If two configurations have the same offset, then there exists a program that moves one to the other.** This arises from the discussion about program length above. Since the two configurations have the same offset invariant, then for each individual ring, there exists a program to transform one to the other, and they both have the same quantity $N_+ - N_-$.

$$\begin{aligned}0 &= \left(\sum \tilde{x}_i^0 - \sum \tilde{x}_i^1\right) - \left(\sum x_i^0 - \sum x_i^1\right)\\ &= \left(\sum \tilde{x}_i^0 - \sum x_i^0\right) - \left(\sum \tilde{x}_i^1 - \sum x_i^1\right)\\ &= (N_+^0 - N_-^0) - (N_+^1 - N_-^1)\end{aligned}$$

Both have a minimal program, so it suffices to take the larger of the two, and rotate the rings so that both programs run and cancellations occur as needed on the ring with the smaller minimal program.

This proof easily extends to any finite pleasant ringset.

Finiteness is required for there to exist a largest minimal program, but what if the ringset isn't pleasant? For a simple counter-example, consider a ringset with two rings of the same size. In this case, the rings are always rotationally in sync, and a single offset invariant is not enough to guarantee that one configuration can be reached from another. Instead, recall our definition of *offset families*. These, which we arrived at from understanding restrictions on index combinations due to rings with non-mutually prime sizes, each require their own offset invariant, as they correspond to completely independent sections of memory on the ring.

However, in practice, we don't work with the domain $\mathbb{Z}$, but rather with some number of bits, where upon reaching the maximum value, it overflows back to zero. From here, we will consider cells with values ranging from 0 to n.

Previously, offset was simply defined as the difference in the sums of a pair of wheels. In this case, however, it is entirely possible to achieve different offset values due to overflow. The actual invariant in this case is offset mod $n + 1$. In this case, when an overflow transition happens (and `+` maps $n \to 0$ and `-` maps $0 \to n$), the offset mod $n + 1$ is still conserved:

$$\begin{aligned}\text{given: } & \sum x_i^1 - \sum x_i^2 && \equiv C \pmod{n+1}\\ \text{then: } & \left(\sum x_i^1 - n\right) - \left(\sum x_i^2 + 1\right)\\ & \equiv C - (n+1) \pmod{n+1}\\ & \equiv C \pmod{n+1}\end{aligned}$$

This completes the forward condition, wherein every program preserves the offset mod $n + 1$.

By now, it should be intuitive that the order of increment (`+`) and decrement (`-`) commands does not matter to the program, as long as the rings are rotated correctly. This fact, combined with the fact that the minimal programs for each wheel must differ by a multiple of $n+1$ in order for the final results to have the same offset mod $n + 1$, we see that we can construct our program by grouping commands in groups of $n + 1$. By rotating the wheels in such a way that the commands are executed in the correct place on certain wheels, while in the same place on others, it is possible to utilize overflow to reach an intermediate configuration that has the same offset (not just the same offset mod $n+1$). From there, we use the earlier result for normal offsets.

In practice, this proves that in order to have complete freedom in assigning values on a pleasant ringset, one cell on each ring except the first must be reserved for the offset, and more cells for unpleasant ones.

Alternatively, a creative solution to the issue of offsets is through the use of the input command (`,`) to manually change the offset on different rings, making different configurations accessible.

3 Conclusion

We reiterate that Ringfuck is not the addition of quality of life features to an existing programming language. It is a new paradigm, a new way of thinking about memory, and a constant reminder of the ordinary person's material insignificance in the circle of life.

It's about coming full circle.

It's about getting jerked around by the circular nature of all things, all while fully coming into the circle.

It's a circle jerk.

Anyways, circular memory also has some natural benefits. For example Ringfuck, in addition to training programmers to be ring fit (Nintendo, 2019), it supports the efficient rounding of numbers (they are already round in memory, what else do you want?), and it is inherently safer than blocks of memory, as demonstrated by Federal Highway Administration Office of Safety (2015) in their work on round-abouts vs. intersections. It is also intuitive to use in most areas of scientific computing, which frequently deal with spherical animals in airless environments, but unfortunately not in orbital mechanics, which deals primarily with ellipses.

It's easy to see why this language will be a massive success. In fact, it's provable! Using only a minor extension to the ZFC axioms of set theory that includes the axiom: "Ringfuck is a massive financial success", we have single-handedly paid for our mortgage. The proof is as follows:

> Ringfuck is a massive financial success because (as per axiom 9) Ringfuck is a massive financial success.

Some might object that this line of reasoning tater-tautological (however tasty). However, we are simply basing our methodology on circular reasoning, which has been popular with philosophers, scientists, and especially politicians since ancient times. It is yet another sign that circular logic, which we hope to bring to the world of programming languages, has been underutilized in favor of more conventional, but less powerful, methods like common sense.

At the end of the day, just remember that if you like something, you should put a ring on it, and that Ringfuck is a wheel great time!

4 Acknowledgements

Special thanks to Spheal, from Pokemon.

5 Disclosure

Sponsored by some company that makes hula hoops or whatever. We disclose that we are directly funded by big circle.

Bibliography

Federal Highway Administration Office of Safety (2015). *Accelerating Roundabout Implementation in the United States* **IV**.

Garland, P. (2015). Arrival.

Hornby, R. (2023). ringfuck interpreter. https://github.com/RyanHornby/ringfuck.

Nintendo (2019). Nintendo ring fit adventure. `https://ringfitadventure.nintendo.com/`.

Perlis, A. J. (1982). Special feature: Epigrams on programming. *SIGPLAN Not.* **17**, 9, 7–13.

Stellman, S. D. (1973). A spherical chicken. *Science* **182**, 4119, 1296–1296.

Thanh, C. H. (2022). br++: A much needed extension to brainfuck. *SIGBOVIK* .

Appendix

Below is a simple “Hello World” program. If you still don’t believe it works feel free to run it through our interpreter (Hornby, 2023).

```
3 4 5
{First load ‘H’ (72 in ascii) into the "0" cell of all the rings}
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+++++++
{Set the "0" cell of ring 2 to ‘E’}
>>>>
---
{Set the "0" cell of ring 3 to ‘L’}
>
++++
{Print all the "0" cells}
<<<<<
.
{Set the "0" cells to ‘LIP’}
++++
{Set the "0" cell of ring 2 to ‘O’}
>>>>
++++++
{Set the "0" cell of ring 3 to ‘ ’}
>
------------------------------------------------
{Print all the "0" cells}
<<<<<
.
{Set the "0" cells to ‘WZ+’}
+++++++++++
{Set the "0" cell of ring 2 to ‘O’}
>>>>
-----------
{Set the "0" cell of ring 3 to ‘R’}
>
+++++++++++++++++++++++++++++++++++++++
{Print all the "0" cells}
<<<<<
.
{Set the "0" cells to ‘LDG’}
-----------
{Set the "0" cell of ring 3 to ‘!’}
>>>>>
--------------------------------------
```

```
{Print all the "0" cells}
<<<<<
.
{Set the "0" cells to ‘ ’, 24, and 245}
--------------------------------------------
{Set the "0" cell of ring 2 to ‘ ’}
>>>>
++++++++
{Set the "0" cell of ring 3 to ‘\n’}
>
+++++++++++++++++++++
{Print all the "0" cells}
<<<<<
.
```