Effective Multi-Threading In Befunge

Zachary Wade

September 19, 2018

Abstract

Befunge is among the most premiere programming languages to have ever been created. With a simple yet powerful feature set, intuitive program flow, and true platform independence, there are few reasons not to use Befunge. However, in a world that has become so obsessed with efficient and fast algorithms, Befunge's single threaded limitations prevent it from being widely adopted by the current generation of computer scientists. In this paper, we will examine multi-threaded Befunge in the context of the newly-minted Befungell language.

1 Befunge: A Background

The original version of Befunge (now known as Befunge-93) was truly a marvel of programming language design. Forgoing standard paradigms like classes, objects, or even types, it instead made use of a truly novel two-dimensional program execution layout. Let us, for a moment, consider the unadulterated genius of this design decision. Not only does it exceed the linear limitations of a standard turing-machine-style programming language, but it frees the developer up to use and reuse code creatively. Want to add a comment? Just route the execution around the text. Want to reuse a portion of code? Just jump into the middle of that area.

Not only is the program execution brilliant, but the very simplicity of program design makes Befunge a revolutionary language. Instead of managing a ton of individual variables, Befunge provides only a single stack – data goes in, data comes out. On top of that, Befunge is a truly dynamic language; it can modify itself as it's running. Few other programming languages have such flexibility. Consider, for instance, Figure 1, a very readable "FizzBuzz" program. As is obvious, execution begins in the top left, and is directed to the right where the main control loop begins. In Befunge, control loops are literal; unlike other more heretical languages, when Befunge loops, its instruction pointer physically moves in circles. As such, we see the code brilliantly model the program's behavior.

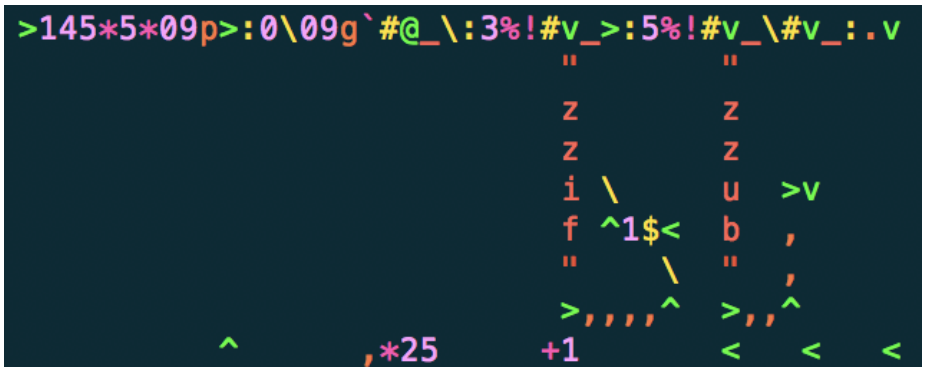

Figure 1: A FizzBuzz Program

1. Not Java
2. Java
3. A Type Theorists' Nightmare
4. Misaligned
5. Dirty Hacks™
6. **Befunge**

Figure 2: Top Languages (githut.info)

2 Single-Threaded Limitations

Given all of these premiere features, one might wonder what prevents Befunge from ascending to the ranks of the top languages. As Figure 2 shows, Befunge is only the 6th most popular language on Github. We speculate that this is due to the major limiting factor that befunge **does not support multi-threading**. In an age of big data and massively parallel computer systems, we find that Befunge's requirement that it operate linearly at all times to be insufficient for the modern world. As such we propose an addendum to the Befunge specification that supports these multi-threaded applications entitled Befungell.

3 Prior Work

Although this may come as a surprise, Befungell is not the first attempt at making a multi-threaded befunge application. In fact, a number of fungeoids have attempted this. However, they all suck.

4 Design Choices

One of the first major design decisions that came with Befungell was its name. We wanted to both pay homage to the language on which it is based, while at the same time encapsulating the raw power of its parallel language structures. To this end, we tried a number of different names. (See Figure 3). However, we settled on Befungell as a concatenation of Befunge and ‖, the international symbol for "parallel." As such, Befungell was born.

From there we had to design the Befungell language extensions. We wanted to treat them a bit like kernel extensions – really annoying to do by hand, but pretty useful if someone else built them for you. Toward this end, we added in two new modes of concurrent operation. One that allows for traditional "fork-join" parallelism and another that provides for more complicated concurrency.

Firstly, we introduced the spawn operator denoted by =. When an instruction pointer enters this block, it immediately hangs. Then, it spawns two new threads and places one instruction pointer at the left of the = sign, and one at the right. Both instruction pointers will be moving away from the spawn operator. Each instruction pointer will operate in their own thread and with a stack copied from the parent process. They can then operate independently until they encounter a termination symbol (@). Once they reach such a symbol, the top value is popped off their stack and pushed onto the stack of the parent process. Then the child thread is terminated. Once both spawned children have been killed, the original spawning thread continues with two values from its children. The original thread is unfrozen and continues moving in the same direction it began. This allows for traditional fork-join operations in Befungell.

We opted for the = sign because of its inherent representation of two parallel lines. In the same way that ‖ in Befungell represents parallelism, so does the = operator. Furthermore, the symmetry of the icon represents the symmetry of the two created threads, which are identical save the duality of their location and initial direction. You can see this in practice in Figure 4

However, for those who desire more control from their threads, we allow for inter thread communication in Befungell via the operations grid. Since this grid is globally readable and writable, we made the grid shared between all threads, and introduced atomic read and write operations so that threads could access the grid without worrying about racing. In addition, we added a single semaphore construct to the language. This is introduced via the new { and } operators. The } operator increments the global semaphore, whereas { pauses the thread until the semaphore is non-zero, then atomically decrements it and continues the current thread.

After significant debate, we chose this syntax to appease those petty C programmers who like to wrap all their code in {Blocks}. Well, now if they want to run concurrent code atomically, all they need to do is wrap the sensitive region in brackets. For an example of atomic printing, see Figure 5.

- Befungelized
- pfunge
- Conc'd Out
- Befungelton Spoonhauer
- Dude like, pthreads in Befunge!
- BeBfefuungnege

Figure 3: Candidate Names

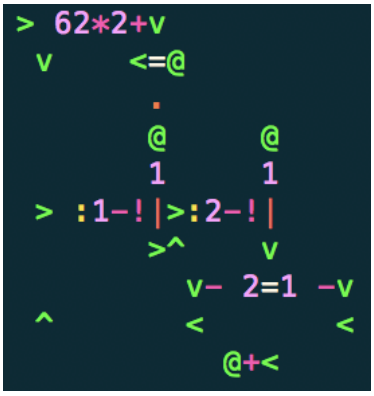

Figure 4: Fibonacci using Fork-Join

Figure 5: Race Free Code

5 Effective Multi-threading In Befunge

Now that we have developed these language constructs, we investigate the techniques for proper multi-threading. For this, we will use a Befungell interpreter written in conjunction with this paper available online at `github.com/zwade/Befungell`.

Already we have shown example programs that make use of these new parallel language constructs. However, other than by being a certified genius like me, one might wonder how to go about designing parallel and concurrent Befungell programs. To this end, we will introduce some elementary techniques that can be combined to form more complex Befungell structures.

The first of these structures is the parallel subroutine. By using a spawn operator, we can compute two pieces of data in parallel, and then have them return to the parent. However, if we only want to have one subroutine start while execution continues normally, one might wonder how we would go about this. One technique is to have the subroutine be executed in a critical (semaphore protected) block. Then, prior to leaving that block, it writes its data to a dedicated square on the grid. Then, the second thread spawned with the spawn operator will continue normal execution, and when the new controller thread needs to read that data value, it will first pass through a critical protected block. This can be visualized in Figure 6.

Another issue one might encounter when writing concurrent Befungell is a limitation imposed by having only a single semaphore – i.e. only being able to introduce one lock at a time. However, it is actually possible to create new locks in Befungell by making use of the atomic grid operations. Say that thread A wants to pause execution until thread B has finished computing some value. We can have thread A spin while it waits, and then have thread B modify the grid at thread A's location to allow As execution to continue. For an example of how this looks in practice, consult Figure 7.

The final structure we will consider is a reader-writer lock. We will only go into a high level overview of how this works, since the underlying structures have already been described, but we may use a combination of the singular (built-in) mutex and a spin mutex to achieve a lock that can be read by many entities at a time, but only written by one. To do this, we will have a reader lock protected by the builtin semaphore, and a writer lock protected by the spin mutex. The actual implementation of this should be immediately apparent and trivial to implement.

6 Conclusion

Well, if you've reached this point in the paper, I must say, thanks Mom. I'm surprised you managed to stick with it this long. Hopefully, this paper has illuminated and elucidated the benefits and potential of multi-threading in Befunge. Furthermore, I hope it that it has shown the true power of Befungell as a language extension. On a slightly more serious note, one might wonder if Befungell has any actual use. One of the things I found while working

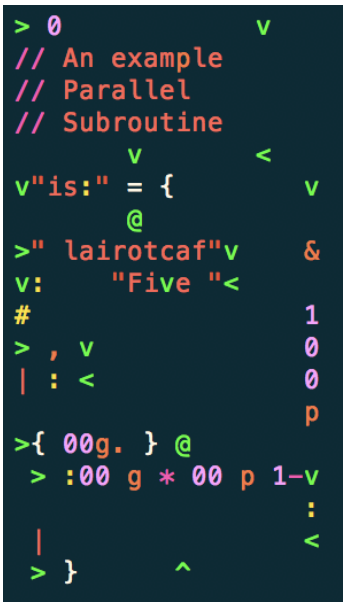

Figure 6: Parallel Subroutine

on this paper is that integrating concurrency and parallelism into a very visual language like Befunge made it far easier to conceptualize what occurs during execution. Befungell, as silly as it is, with sufficient visual overhaul could make an interesting and potentially useful language for introducing more difficult programming concepts to young students. Because of its strong analogue to the real world, children may find it easier to transform their ideas into an executable program. It may be worth investigating this further, and seeing how it could be applied as an educational tool.

Finally, in conclusion, Befunge Good, Befungell Gooder^{TM}.

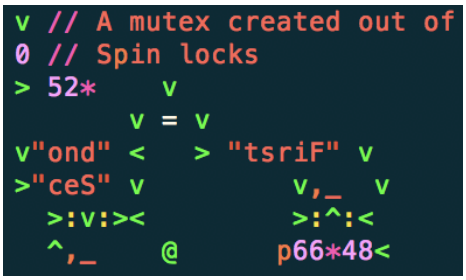

Figure 7: Semaphore-Free Lock

References

[1] Matthew Savage. "Going Bananas: Modeling Chaos Theory with Unexpected Behavior in C", Carnegie Mellon: SIGBOVIK Press, 2018.

[2] My 15-312 TA. *Why Every Language is Terrible*, Carnegie Mellon: Recitation Notes, 2017.

[3] vsync. "vsync's Funge Stuff." Internet: quadium.net/funge, January 1, 1993, [Epilepsy Warning]

[4] Carlo Zapponi. "Githut - Programming Languages and Github." Internet: githut.info, 2014, [March 12, 2017]

[5] Zachary Wade, "What do you mean I can't do CodeForces in Befunge!." Rant, 2016

[6] David Lanman, "Do it: Why you should write a paper about Concurrent Befunge." Facebook Messenger, March 1st 2017

[7] Harry Qandyqorn Bovik, "An Investigation into the Paranormal History of Python." New York: Fictional Press, 1993