Crazy Quines

Maxwell Tang

Due 11 May 2024

Note: This writeup doesn't nessecarily cover everything I did for this project. Rather, it just ties the highlights into a single narrative. The source code containing everything I did for this project is at
`https://github.com/maxwell3025/H250-Project`

Okay, so we've just received our first message from the great unknown, and it looks like the strangers from another world have something to share with us! After decoding the message we wait to see what it is.

Maybe it's a polynomial time solution for 3-SAT.
Maybe it's a proof of the Goldbach conjecture.
Maybe it's a computer virus that destroys our planet.

It's a TypeScript interpreter ...

While we could sit around and ponder the implications of this, it would be rude to leave the aliens on waiting for a response.

To show off our intelligence, we can send a computer program back to them, and since they sent us a Typescript runtime, it would be natural to send a Typescript program back. In addition, we should try to send something that they find interesting, which is a much harder task than it initially seems. For example, they might not reason using the same mathematical framework we do, and they probably don't have a use for tools that are useful to us, and they definitely don't know English or any other languages from earth. However, they *did* send us a Typescript runtime, so we can be reasonably sure that they understand computer science, which means that a program that's interesting computationally would probably be interesting to them too. Perhaps we send them a quine!

Quine in Typescript

Mathematicians here on Earth have proved that we can create quines in any programming language[Mad], so we have a starting point for creating our message. If we follow the schema of starting with a template for the source code, followed by code for decoding that source code, then we can pretty easily create a quine!

../typescript–quines/shortenedQuine.ts

```
1 const data="const data=\"xy\"\nconsole.log(data.replace
    (/[x][y]/,JSON.stringify(data)))"
2 console.log(data.replace(/[x][y]/,JSON.stringify(data)))
```

Radiation-Hardened Quine in Typescript

The problem is, though, that these guys are thousands of light-years away! There's no way that our signal is getting to them intact. What we need is a quine that can survive being corrupted. As a start, we can try to create a "radiation-hardened quine" — a quine that prints out its intact source code after any single character deletion.

Existing Radiation-Hardened Quines

To begin, it makes sense to look at an example in another language, which we can find on the Code Golf Stack Exchange[jri16]. This script contains a few interesting moving parts that can be ported over to TypeScript. The most significant feature of the script is the 2 identical `setTimeout` calls. The poster explains that `setTimeout` is used to interpret code so that it exits silently when it hits an error. Since it is copied, at least one of the `setTimeout` calls is intact, and will print the intact source code correctly.

In addition, the script begins with a "guard statement" that reassigns any typos to a function that does nothing. This also plays a part in ensuring that the mutation doesn't break the script. However, getting this part to work correctly relies on unscoped declarations, which don't exist in TypeScript. This leads us into the key issue with making a radiation-hardened quine in TypeScript.

Unfortunately, every valid TypeScript program must start with a keyword such as `const`, `let`, `function`, etc. Since the keywords in TypeScript are not lexically close enough to allow for clever tricks, any script can be made invalid by changing the first keyword. Indeed, similar reasoning shows that most languages, like C, C++, Rust, Java, etc. have keywords as their Achilles' heels. This, however, shouldn't lead us to lose hope. The aliens probably understand that signals can be corrupted when traveling hundreds of light years, so we just need to maximize the probability that out script will run correctly. Thus, we can loosen our requirements and see what can be done. For a start, we can still use the tricks outlined above to make our script valid under *most* deletions, so that we will still probably get our message across. In addition, we should also make sure that the script doesn't output the wrong text. Instead, it should just crash when it can't recover, since we definitely don't want the program to mislead the aliens into thinking it's functioning correctly.

When we run our original quine though a testing script, we get 77 mutants which output the wrong text, and 0 recoverable mutations. This is terrible!

Now, we can try to write a quine which recognizes any errors within itself(see `typescript-quines/harderQuine.ts`). This uses a checksum in order to

validate all of the strings in the code, and manually exits with an error if it doesn't match. When we test this, we get 325 mutations that are recoverable, 0 incorrect outputs, and 1382 crashes. This is better, but most of the mutations just skip, which doesn't bode well for our odds of getting this message successfully sending.

Finally, if we use redundancy, as mentioned in the example, we can now recover from mutations, rather than just recognizing them. In the final script, I use three redundant copies of the data string to allow for any incorrect strings to be thrown out. In this final script, we get 748 recoverable mutations, and only 194 crashing mutations. This is a much better ratio than we got before, and this was the best I was able to do.

Conclusion

With our final program ready, we can send our message out into space!

References

[jri16] jrich. *Javascript (ES6)*. Stack Exchange. 2016. URL: `https://codegolf.stackexchange.com/questions/57257/radiation-hardened-quine/#answer-73753`.

[Mad] David Madore. *Quines (self-replicating programs)*. Web. URL: `http://www.madore.org/~david/computers/quine.html`.