

Progress report in Pen programming language

June 19th, 2022

[@raviqqe](#)

Agenda

- Progress report
 - Relaxed atomic operations in reference counting
- Next plans

Progress report

Relaxed atomic operations in reference counting

- Reference counting can use relaxed atomic or non-atomic operations sometimes.
 - For only references never shared by multiple threads
- Part of [the Perceus reference counting algorithm](#)

Benchmark

- Conway's game of life
- Size: 20 x 40
- Iterations: 100

```
> hyperfine -w 3 ~/app-old ~/app-new
Benchmark 1: /home/raviqqe/app-old
  Time (mean ± σ):      1.712 s ±  0.011 s    [User: 1.658 s, System: 0.012 s]
  Range (min ... max):  1.694 s ...  1.727 s    10 runs

Benchmark 2: /home/raviqqe/app-new
  Time (mean ± σ):      1.175 s ±  0.021 s    [User: 1.124 s, System: 0.008 s]
  Range (min ... max):  1.152 s ...  1.229 s    10 runs

Summary
  '/home/raviqqe/app-new' ran
    1.46 ± 0.03 times faster than '/home/raviqqe/app-old'
```

Record update benchmark

- The previous result had several performance bugs!
 - Inefficient map literal compilation
 - Non-unique references

Configuration

- Hash map initialization with many entries
- A number of entries: 100,000
- Key type: 64-bit floating point number

Results

- Pen is 6 ~ 7 times slower than Rust currently...

Pen

```
> hyperfine -w 3 ./app
Benchmark 1: ./app
  Time (mean ± σ):      274.0 ms ±   2.7 ms    [User: 206.7 ms, System: 16.8 ms]
  Range (min ... max):  269.6 ms ... 279.3 ms    10 runs
```

im-rs in Rust

- `HashMap::insert(&mut self, k: K, v: V) -> Option<V>`

```
test hashmap_insert_mut_100000 ... bench: 34,869,571 ns/iter (+/- 4,337,627)
```

Next plans

- Reference counting optimization
 - Unboxing small records [#671](#)
 - Other basic optimizations
- Proper C calling convention in FFI [#444](#)
 - Compiling to MLIR?

Summary

- Progress
 - Relaxed atomic operations in reference counting
- Next plans
 - Record unboxing

Appendix

Record update benchmark

No uniqueness check

```
> for _ in $(seq 5); do time ./app; done
./app 7.98s user 0.26s system 99% cpu 8.302 total
./app 7.70s user 0.24s system 99% cpu 7.991 total
./app 7.71s user 0.29s system 99% cpu 8.052 total
./app 7.76s user 0.31s system 99% cpu 8.117 total
./app 8.10s user 0.26s system 99% cpu 8.423 total
```

Acquire ordering

```
> for _ in $(seq 5); do time ./app; done
./app 7.68s user 0.28s system 99% cpu 8.019 total
./app 7.57s user 0.32s system 99% cpu 7.950 total
./app 7.63s user 0.22s system 99% cpu 7.905 total
./app 7.58s user 0.26s system 99% cpu 7.899 total
./app 7.59s user 0.27s system 99% cpu 7.933 total
```

Relaxed (buggy) ordering

```
> for _ in $(seq 5); do time ./app; done
./app 7.46s user 0.30s system 99% cpu 7.817 total
./app 7.41s user 0.24s system 99% cpu 7.703 total
./app 7.51s user 0.26s system 99% cpu 7.823 total
./app 7.47s user 0.26s system 99% cpu 7.775 total
./app 7.42s user 0.26s system 99% cpu 7.734 total
```

Relaxed (correct) ordering

```
> for _ in $(seq 5); do time ./app; done
./app 7.60s user 0.26s system 99% cpu 7.930 total
./app 7.50s user 0.29s system 99% cpu 7.860 total
./app 7.60s user 0.27s system 99% cpu 7.940 total
./app 7.56s user 0.25s system 99% cpu 7.865 total
./app 7.55s user 0.27s system 99% cpu 7.878 total
```

Game of life benchmark

- Relaxed atomic operations for thunks

Before:

```
> hyperfine ./app
Benchmark 1: ./app
  Time (mean  $\pm$   $\sigma$ ):    11.637 s  $\pm$   0.191 s    [User: 11.588 s, System: 0.094 s]
  Range (min ... max):    11.311 s ... 11.992 s    10 runs
```

After:

```
> hyperfine ./app
Benchmark 1: ./app
  Time (mean  $\pm$   $\sigma$ ):    11.891 s  $\pm$   0.146 s    [User: 11.822 s, System: 0.109 s]
  Range (min ... max):    11.614 s ... 12.097 s    10 runs
```