3/7/2023

When Zig is safer and faster than Rust

There are endless debates online about Rust vs. Zig, this post explores a side of the argument I don't think is mentioned enough.

Intro / TLDR

I was intrigued to learn that the Roc language rewrote their standard library from Rust to Zig. What made Zig the better option?

They wrote that they were using a lot of unsafe Rust and it was getting in their way. They also mentioned that Zig had *"more tools for working in a memory-unsafe environment, such as reporting memory leaks in tests"*, making the overall process much better.

So is Zig a better alternative to writing unsafe Rust?

I wanted to test this myself and see how hard unsafe Rust would be by building a project that required a substantial amount of unsafe code.

Then I would re-write the project in Zig to see if would be easier/better.

After I finished both versions, I found that the Zig implementation was safer, faster, and easier to write. I'll share a bit about building both and what I learned.

What we're building

We'll be writing a bytecode interpreter (or VM) for a programming language that uses mark-sweep *garbage collection*.

The garbage collection is the important part, it's hard to make it work *and* be fast *and* be safe because its fundamentally a problem that doesn't play nicely with the borrow checker.

There are two ways I can think of to do it in safe Rust: using reference-counting[1] and using arenas+handles[2], but both seem to be slower than a traditional mark/sweep approach[3].

The specific bytecode interpreter implementation is from one of my favorite books: Crafting Interpreters. In particular, it's a stack-based VM for a language that supports functions, closures, classes/instances, etc.

You can view the code for this here.

The unsafe Rust implementation

Overall, the experience was not great, especially compared to the joy of writing regular safe Rust. I'll hone in on why that is exactly.

Unsafe Rust is hard

Unsafe Rust is hard. A lot harder than C, this is because unsafe Rust has a lot of nuanced rules about undefined behaviour (UB) — thanks to the borrow checker — that make it easy to perniciously break things and introduce bugs.

This is because the compiler makes optimizations assuming you're following its ownership rules. But if you break them, thats considered undefined behaviour, and the compiler chugs along, applying those same optimizations and potentially transforming your code into something dangerous.

To make matters worse, Rust doesn't fully know what behaviour is considered undefined[4], so you could be writing code that exhibits undefined behaviour without even knowing it.

One way to alleviate this is to use Miri, it's an interpreter for Rust's mid-level intermediate representation that can detect undefined behaviour.

It does its job, but it comes with its own warts[5].

Here's Miri showing me undefined behaviour:

```
running 1 test
test test::string ... error: Undefined Behavior: no item granting write access for deallocation to tag <288471> at alloc109018 found in borrow stack
   --> /Users/zackradisic/.rustup/toolchains/nightly-2022-10-08-aarch64-apple-darwin/lib/rustlib/src/rust/library/alloc/src/alloc.rs:113:14
    |
113 |     unsafe { __rust_dealloc(ptr, layout.size(), layout.align()) }
    |              ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ no item granting write access for deallocation to tag <288471> at alloc109018 found in bor
row stack
    |
    = help: this indicates a potential bug in the program: it performed an invalid operation, but the Stacked Borrows rules it violated are still experimental
    = help: see https://github.com/rust-lang/unsafe-code-guidelines/blob/master/wip/stacked-borrows.md for further information
help: <288471> was created by a SharedReadWrite retag at offsets [0x0..0x1]
   --> src/obj.rs:63:40
    |
63  |                         alloc::dealloc((*obj_str).chars.as_mut(), layout);
    |                                        ^^^^^^^^^^^^^^^^^^^^^^^^^
    = note: BACKTRACE:
    = note: inside `std::alloc::dealloc` at /Users/zackradisic/.rustup/toolchains/nightly-2022-10-08-aarch64-apple-darwin/lib/rustlib/src/rust/library/alloc/src
/alloc.rs:113:14
note: inside `obj::Obj::free` at src/obj.rs:63:25
   --> src/obj.rs:63:25
    |
63  |                         alloc::dealloc((*obj_str).chars.as_mut(), layout);
    |                         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

The most challenging part: Aliasing rules

The most challenging source of undefined behaviour I ran into was related to Rust's aliasing rules.

As mentioned before, Rust leverages its borrowing/ownership rules to make compiler optimizations. If you break these rules, you get undefined behaviour.

So how can I even write this?

The trick is to use **raw pointers**. They don't have the same borrowing constraints as regular Rust **references**, allowing you to side-step the borrow checker.

For example, you can have as many mutable (`*mut T`) or immutable (`*const T`) raw pointers as you want.

Then the solution should be easy, right? Not quite.

If you turn a **raw pointer** into a **reference** (`&mut T` or `&T`), you have to make sure it **abides by the rules for that type of reference** for the duration of its lifetime. For example a mutable reference cannot exist while other mutable/immutable references exist too.

```
fn do_something(value: *mut Foo) {
    // Turn the raw pointer into a mutable reference
    let value_mut_ref: &mut Foo = value.as_mut().unwrap();

    // If I create another ref (mutable or immutable) while the above ref
    // is alive, that's undefined behaviour!!

    // Undefined behaviour!
    let value_ref: &Foo = value.as_ref().unwrap();
}
```

This is really easy to violate. You might make a mutable reference to some data, call some functions, and then like 10 layers deep into the call stack one function might make an immutable reference to that same data, and now you have undefined behaviour. Woo!

Okay, so what if we just avoid making references entirely and only use raw pointers?

Well, the issue with that is raw pointers don't have the same ergonomics as references. Firstly, you can't have associated functions that take the `self` as a raw pointer:

```
struct Class {
    /* fields... */
}
```

```
impl Class {
    // Regular associated function
    fn clear_methods(&mut self) {
        /* ... */
    }

    fn clear_methods_raw(class: *mut Class) {
        /* ... */
    }
}

unsafe fn test(class: *mut Class) {
    let class_mut_ref: &mut Class = class.as_mut().unwrap();
    // This syntax is nice and ergonomic
    class_mut_ref.clear_methods();

    // But with raw pointers you'll have to just call the function like in C
    Class::clear_methods_raw(class);
}
```

Next, there isn't any nice pointer derefence syntax like C's `ptr->field`. The code is littered with ugly `(*ptr).field` everywhere, it's disgusting.

It really sucks when you have to chain dereferences. Here are two monstrosities I found in the code:

```
// The way to make these readable is to create a variable for each dereference,
// but that's annoying so in some places I got lazy.

// ew
(*(*closure.as_ptr()).upvalues.as_ptr().offset(i as isize)) = upvalue;

// ewwwwww
let name = (*(*(*ptr.cast::<ObjBoundMethod>().as_ptr()).method.as_ptr())
    .function
    .as_ptr()).name;
```

Another problem was working with arrays.

If I have a raw pointer to an array of data (`*mut T`), I can turn it into a slice `&mut [T]`, and I get to use a `for ... in` loop on it or any of the handy iterators (`.for_each()`, `.map()`, etc.).

But turning it into a `&mut [T]` basically makes a reference to all the data in the array, so it

makes it really easy to violate Rust's aliasing rules yet again.

```
unsafe fn do_stuff_with_array(values: *mut Value, len: usize) {
    let values: &mut [Value] = std::slice::from_raw_parts_mut(values, k);
    // I can use the ergonomics of iterators!
    for val in values {
        /* ... */
    }
    // I just have to make sure none of the Values are turned
    // into references (mutable or immutable) while the above slice is active...
}
```

Again, the solution is to avoid making references, so in some places I ended up writing regular C-style for-loops on the array raw ptr. But raw pointers suck in comparison to Rust's slices because you can't index them and you don't get out-of-bounds checking.

```
pub struct Closure {
    upvalues: *mut Upvalue
}

unsafe fn cant_index_raw_pointer(closure: *mut Closure) {
    // Can't do this:
    let second_value = (*closure.upvalues)[1];

    // Have to do this, and no out-of-bounds checking
    let value = *(*closure).upvalues.offset(1);
}
```

Miri uses the Stacked Borrows model so it can detect UB related to these aliasing rules mentioned above, but fixing them was challenging.

It felt a lot like when I was learning Rust, there are these rules that exist but I don't have a solid mental model of them.

I would have to figure out why what I was doing was wrong, then experiment to find a way to fix it. Also note that the feedback loop is way slower because there's no LSP in my code editor guiding me, I have to recompile the program each time and see Miri's output.

This really ruined the experience for me. By the end of it I wasn't really writing Rust, just this weird half-Rust half-C language that was way more error prone and delicate.

The Zig implementation

After spending a lot of time practicing the dark arts in Rust, I was excited to leave unsafe Rust and learn Zig and start rewriting the project in it.

Apart from not having crazy UB like in unsafe Rust, Zig is a language that understands that you are going to be doing memory-unsafe things, so its designed and optimized around making that experience much better and less error prone. These were some key things that helped:

Explicit allocation strategies

In Zig, any function that allocates memory must be passed an `Allocator`.

This was amazing because I made the garbage collector a custom `Allocator`. Each allocation/deallocation tracked how many bytes were allocated and triggered gargbage collection if needed.

```
const GC = struct {
    // the allocator we wrap over that does
    // the heavy lifting
    inner_allocator: Allocator

    bytes_allocated: usize

    // and other fields...

    // `alloc`, `resize`, and `free` are required functions
    // to implement an Allocator
    fn alloc(self: *GC, len: usize, ptr_align: u29, len_align: u29, ret_addr: usiz
        // let the inner allocator do the work
        const bytes = try self.inner_allocator.rawAlloc(len, ptr_align, len_align,

        // keep track of how much we allocated
        self.bytes_allocated += bytes.len;

        // collect if we exceed the heap growth factor
        try self.collect_if_needed();

        return bytes;
    }
};
```

What I love about this design choice of Zig's is it makes it frictionless and idiomatic to use different allocation strategies that are optimal for your use-case.

For example, if you know some allocations have a similar and finite lifetime, you can use a super fast bump arena allocation (or linear arena allocation) to speed up your program. This stuff exists in Rust, but it's not as nice as in Zig[6].

A special allocator which detects memory bugs

When used, it detects use-after-frees and double-frees. It prints a nice stack trace of when/where the data was allocated, freed, and used.

```
// All you have to do is this
const alloc = std.heap.GeneralPurposeAllocator(.{
    .retain_metadata = true,
}){};
```

This in particular was a life-saver during development.

Non-null pointers by default

Most memory safety bugs are null pointer dereferences and out-of-bounds array indexing.

Rust's raw pointer types are nullable by default and have no null-pointer dereference checking. There is a NonNull<T> pointer type that gives you more safety. I'm not sure why it's not the default, since Rust's references are non-nullable.

Zig pointers, by default, are non-null and you opt-in to nullability using the `?` syntax (e.g. `?*Value`). And of course, you get null pointer dereference checking enabled by default too.

I much prefer this, it makes the default choice the safest.

Pointers and slices

Zig understands you are going to be working with pointers, so it makes that experience great.

A big problem with the unsafe Rust version was that raw pointers had terrible ergonomics. The syntax for dereferencing was awful, and I couldn't index raw ptr arrays using the `slice[idx]` syntax.

Zig's pointers have the same ergonomics as Rust references, namely that the dot operator doubles as pointer dereferencing:

```
fn do_stuff(closure: *Closure) {
    // This dereferences `closure` to get the
    // `upvalues field`
    const upvalues = closure.upvalues;
}
```

Zig also has some additional pointer types that help you disambiguate "pointers to a single value" from "pointers to an array":

```
const STACK_TOP = 256;
const VM = struct {
    // pointer to unknown number of items
    stack_top: [*]Value,
    // like a rust slice:
    // contains a [*]Value + length
    // has bounds checking too
    stack: []Value,
    // alternative to slices when
    // N is a comptime known constant
    stack_alt: *[STACK_TOP]Value
};
```

These support indexing with the `array[idx]` syntax, while regular pointers (`*T`) don't, which is really great for safety.

The cool part is that it's very easy to convert between the different pointer types:

```
fn conversion_example(chars: [*]u8, len: u8) []Value {
    // Converting to a []T is easy:
    var slice: []const u8 = chars[0..len];

    // And back
    var ptr: [*]u8 = @ptrCast([*]u8, &slice[0]);
}
```

"Traditional" pointers, like the ones you'd see in C/C++, are very error prone.

Rust solves this by adding a facade layer over pointers: its reference types (`&T` or `&mut T`). But unfortunately for Rust, its raw pointers still have the same issues as in C/C++.

Zig solves this by simply removing a lot of the footguns from pointers and adding additional guard rails.

Benchmark Results

Now it's time for what everyone wants to see, the benchmarks[7].

Results

Here are a few pics of the results, if you want to look at the specific code check out the repo.

1. 35th fibonacci number

```
make bench BENCH_PROG=fib

hyperfine --warmup 5 './zlox/zig-out/bin/zlox ./benchmarks/fib.lox' './loxide/target/release/loxide ./benchmarks/fib.lox'
Benchmark 1: ./zlox/zig-out/bin/zlox ./benchmarks/fib.lox
  Time (mean ± σ):      1.077 s ±  0.037 s    [User: 1.055 s, System: 0.009 s]
  Range (min …max):    1.053 s … 1.172 s    10 runs

  Warning: The first benchmarking run for this command was significantly slower than the rest (1.172 s). This could be caused by (filesystem) caches that were no
t filled until after the first run. You should consider using the '--warmup' option to fill those caches before the actual benchmark. Alternatively, use the '--p
repare' option to clear the caches before each timing run.

Benchmark 2: ./loxide/target/release/loxide ./benchmarks/fib.lox
  Time (mean ± σ):      1.657 s ±  0.019 s    [User: 1.641 s, System: 0.012 s]
  Range (min …max):    1.634 s … 1.688 s    10 runs

Summary
  './zlox/zig-out/bin/zlox ./benchmarks/fib.lox' ran
    1.54 ± 0.06 times faster than './loxide/target/release/loxide ./benchmarks/fib.lox'
```

Zig is 1.54 ± 0.06 times faster than Rust.

2. method calling stress test

```
make bench BENCH_PROG=method_call

hyperfine --warmup 5 './zlox/zig-out/bin/zlox ./benchmarks/method_call.lox' './loxide/target/release/loxide ./benchmarks/method_call.lox'
Benchmark 1: ./zlox/zig-out/bin/zlox ./benchmarks/method_call.lox
  Time (mean ± σ):     186.4 ms ±   3.3 ms    [User: 184.0 ms, System: 1.6 ms]
  Range (min …max):   181.5 ms …191.7 ms    15 runs

Benchmark 2: ./loxide/target/release/loxide ./benchmarks/method_call.lox
  Time (mean ± σ):     328.0 ms ±   4.0 ms    [User: 324.3 ms, System: 2.6 ms]
  Range (min …max):   323.3 ms …334.0 ms    10 runs

Summary
  './zlox/zig-out/bin/zlox ./benchmarks/method_call.lox' ran
    1.76 ± 0.04 times faster than './loxide/target/release/loxide ./benchmarks/method_call.lox'
```

Zig is 1.76 ± 0.04 times faster than Rust.

3. string equality stress test

```
make bench BENCH_PROG=string_equality

hyperfine --warmup 5 './zlox/zig-out/bin/zlox ./benchmarks/string_equality.lox' './loxide/target/release/loxide ./benchmarks/string_equality.lox'
Benchmark 1: ./zlox/zig-out/bin/zlox ./benchmarks/string_equality.lox
  Time (mean ± σ):      94.1 ms ±   3.0 ms    [User: 92.7 ms, System: 0.5 ms]
  Range (min …max):    90.9 ms …102.3 ms    27 runs

Benchmark 2: ./loxide/target/release/loxide ./benchmarks/string_equality.lox
  Time (mean ± σ):     147.1 ms ±   2.2 ms    [User: 145.5 ms, System: 0.5 ms]
  Range (min …max):   143.2 ms …151.0 ms    19 runs

Summary
  './zlox/zig-out/bin/zlox ./benchmarks/string_equality.lox' ran
    1.56 ± 0.05 times faster than './loxide/target/release/loxide ./benchmarks/string_equality.lox'
```

Zig is 1.56 ± 0.05 times faster than Rust.

4. zoo

```
make bench BENCH_PROG=zoo

hyperfine --warmup 5 './zlox/zig-out/bin/zlox ./benchmarks/zoo.lox' './loxide/target/release/loxide ./benchmarks/zoo.lox'
Benchmark 1: ./zlox/zig-out/bin/zlox ./benchmarks/zoo.lox
  Time (mean ± σ):     329.6 ms ±   2.5 ms    [User: 323.8 ms, System: 2.5 ms]
  Range (min …max):   326.5 ms …333.9 ms    10 runs

Benchmark 2: ./loxide/target/release/loxide ./benchmarks/zoo.lox
  Time (mean ± σ):     571.3 ms ±  12.6 ms    [User: 561.5 ms, System: 5.0 ms]
  Range (min …max):   559.2 ms …602.3 ms    10 runs

Summary
  './zlox/zig-out/bin/zlox ./benchmarks/zoo.lox' ran
    1.73 ± 0.04 times faster than './loxide/target/release/loxide ./benchmarks/zoo.lox'
```

Zig is 1.73 ± 0.04 times faster than Rust.

Why was Zig faster?

From the benchmarks, the Zig implementation was around 1.56-1.76x faster than Rust. Why is that?

I tried profiling the two, but I didn't get any useful information out of that.

I think the reason is that in the Rust version there were a few places where I was using indices instead of pointers (for the stack, next instruction, call frame stack, some places for upvalues) because it was too complicated to get them to use pointers and satisfy Miri's UB checks.

The Zig version uses pointers for those places. So super common operations like manipulating the top of the stack, getting the next instruction, or the current call frame were a simple pointer derefence — while Rust had to do pointer arithmetic.

Keep in mind these operations are happening super frequently in the VM, so these extra ops add up.

I tried refactoring the Rust version, but each time I tried it would take a long time and wouldn't satisfy Miri's checks so I gave up after a bit.

If anyone has another idea for why the Rust version is slower or has any ideas on how to speed it up let me know!

Conclusion

Writing a substantial amount of unsafe Rust really sucks the beauty out of the language. I felt like I was either tiptoeing through this broken glass of undefined behaviour, or I was writing in this weird half-Rust/half-C mutated abomination of a language.

The whole point of Rust is to use the borrow checker, but when you frequently need to do something the borrow checker doesn't like... should you really be using the language?

That aspect of Rust doesn't seem to be talked about enough when people compare Rust vs. Zig, and definitely should be considered if you're going to be doing memory-unsafe things for performance.

As someone who loves Rust, I'm definitely going to explore using Zig more for projects. I like the idea of being able choose different memory allocation strategies and the performance opportunities that brings.

While I was writing the Zig version, I took comprehensive notes on my thoughts and feelings learning and using the language. I'll clean that up and post that as separate blog post.

Resources

This 3-part series "What Every C Programmar Should Know About Undefined Behavior" by

LLVM's Chris Lattner, talks more about how compiler optimizations can cause code with undefined behaviour to become unsound.

This section from the Rustonomicon gives an example of breaking Rust's aliasing rules and how a compiler optimization might cause problems.

A post I read 2 years ago by Manuel Cerón inspired the idea for this post.

Checkout Crafting Interpreters, the two interpreters are based on the one from this book.

Here's the repo containing the code for the Rust and Zig interpreters.

Thanks to Caleb, Danny, and Phil Eaton for proofreading this post for me.

Footnotes

[1] Ref-counted pointers have some performance penalties:

- λx. Adding/dropping references wastes precious CPU cycles by having to increment/decrement the reference count.
- λx. The reference count is heap allocated so you blow cache lines when adding/dropping references and thus suffer an additional performance cost
- λx. Reference counting doesn't play nicely with cycles, you need to also implement some mechanism to detect self-referential pointers and manually GC them yourself.

So at the end you have to incur the cost of maintaining reference counts, and will also need to strap on a mini-GC to detect cyclic data.

The only Rust project I know that has implemented this hybrid "Reference-counted GC" design is Goscript, which itself is based on CPython's design.

But that's not to say reference-counting based GC is always slow. Perceus is a ref-counting based GC design for FP languages that leverages the fact that data types in FP langs are immutable, to achieve way less GC overhead and execution time. In some cases it outperforms C++! This strategy is used by the languages Koka and Roc.

[2] The strategy with arenas+handles is to allocate objects into an "arena" (typically a Vec) and instead of pointers/references you use "handles" (typically indices into the Vec). This lets you sidestep some of Rust's borrowing rules.

There is a performance cost because indexing into the arena is slower than a pointer dereference. Instead of simply dereferencing a pointer, you invalidate cache lines to get

the pointer to the beginning of the arena and then you have to do pointer arithmetic. Also it doesn't completely solve your borrow checker problems, sometimes you'll have to make unnecessary clones or do other unnecessary work to satisfy the compiler.

I started an implementation of this strategy here but didn't finish it. It uses a forked version of generational-arena that modifies the handle type so it has a CPU cache-friendlier representation that is 8-bytes wide. It also makes sure the handle is non-zero and wrapts it in a `NonZeroUsize` so Rust's null-pointer optimization applies.

3 This isn't a 100% backed claim. It's general knowledge that reference counting tends to be slower than GC. And as mentioned in the above footnote for the arena strategy, indexing seems like it would be slower than pointer dereference.

But I haven't actually tested this. Most GCs are fast because they use cool optimized techniques like generational garbage collection. We're just building a simple mark/sweep collector, and I don't have time to make 4 different versions to comprehensively compare each strategy.

Keep in mind I'm also doing this for fun to try see how unsafe Rust fares against Zig.

4 The best all-in-one resource I've found is this page from the Rust reference book, but it has a pretty big warning sign:

> *The following list is not exhaustive. There is no formal model of Rust's semantics for what is and is not allowed in unsafe code, so there may be more behavior considered unsafe. The following list is just what we know for sure is undefined behavior.*

5 Miri can detect a lot of UB, even things like use-after-free, but I still had some frustrations with it:

- $\lambda x.$ **It's slow**

  The interpreter isn't exactly fast, from what I've observed it's more than 400x slower. Regular Rust can run the tests I wrote in less than a second, but Miri takes several minutes.

- $\lambda x.$ **It doesn't work with everything**

  I tried to use the mimalloc allocator to speed up allocations, but Miri can't handle foreign functions (the mimalloc crate calls upon the C library).

  This is pretty bad because it means if you use an FFI library you can't test for

undefined behaviour.

$\lambda x.$ **A crate you use might have UB**

If you use a crate in your Rust program, Miri will also panic if that crate has some UB. This sucks because there's no way to configure it to skip over the crate, so you either have to fork and patch the UB yourself, or raise an issue with the authors of the crates and hopefully they fix it.

This happened to me once on another project and I waited a day for it to get fixed, then when it was finally fixed I immediately ran into another source of UB from another crate and gave up.

6

My favorite arena allocator in Rust is bumpalo, which is pretty great. But the Rust language as a whole isn't really designed around custom allocators (yet, it's still a WIP).

If you want to bump-allocate a `Vec<T>` in Rust, you have to use bumpalo's custom `Vec` type, which is literally a copy-paste of the standard library's version modified to allocate with bumpalo's arena type. If you want to support other collections, for example `BTreeMap<K, V>`, you're going to have to copy-paste and modify it from Rust's std library.

In contrast, Zig is designed with custom allocators in mind. You can create an `ArrayList<T>` and give it an allocator, and it will store and use that allocator. If you don't want to incur the cost of the `ArrayList<T>` taking up an additional 8-bytes for the allocator (perhaps if you have many ArrayList's and want a more CPU cache-friendly design), you can create an `ArrayListUnmanaged<T>` and only pass the allocator when pushing or popping from the array list.

7

Systems-level languages like Zig/Rust/C/C++ generally have comparable performance. Zig and Rust in particular use the LLVM backend by default to compile to machine code, both give you enough control over the memory layout of your data to write fast programs, and they both don't have a "runtime" in the traditional sense of the word.

So, I think a better metric than performance is the developer experience. Was the effort I put into Rust worth it over Zig, or vice-versa?

In this particular case, even if my Rust implementation was faster, I'd say no. It was a major pain dealing with all the nuances of unsafe Rust.