Assorted thoughts on zig (and rust)

I've been using zig for ~4 months worth of side projects, including a toy text editor and an interpreter for a relational language. I've written ~10kloc.

That's not nearly enough time to form a coherent informed opinion. So instead here is an incoherent assortment of thoughts and experiences, in no particular order :)

This is not meant to be an introduction to zig - check out the excellent language docs or the new ziglearn.org instead. I'll try to focus instead on things that are not immediately obvious from reading intro material.

The obvious point of comparison is to rust. For context, I've been using rust since 2015. Mostly in research positions writing throwaway code, but also ~14 months working on a commercial database which is ~100kloc.

Zig is dramatically simpler than rust. It took a few days before I felt proficient vs a month or more for rust.

Most of this difference is **not** related to lifetimes. Rust has patterns, traits, dyn, modules, declarative macros, procedural macros, derive, associated types, annotations, cfg, cargo features, turbofish, autoderefencing, deref coercion etc. I encountered most of these in the first week. Just understanding how they all work is a significant time investment, let alone learning when to use each and how they affect the available design space.

I still haven't internalized the full rule-set of rust enough to be able predict whether a design in my head will successfully compile. I don't remember the order in which methods are resolved during autoderefencing, or how module visibility works, or how the type system determines if one impl might overlap another or be an orphan. There are frequent moments where I know what I want the machine to do but struggle to encode it into traits and lifetimes.

Zig manages to provide many of the same features with a single mechanism - compile-time execution of regular zig code. This comes will all kinds of pros and cons, but one large and important pro is that I already know how to write regular code so it's easy for me to just write down the thing that I want to happen.

One of the key differences between zig and rust is that when writing a generic function, rust will prove that the function is type-safe for every possible value of the generic parameters. Zig will prove that the function is type-safe only for each parameter that you actually call the function with.

On the one hand, this allows zig to make use of arbitrary compile-time logic where rust has to restrict itself to structured systems (traits etc) about which it can form general proofs. This in turn allows zig a great deal of expressive power and also massively simplifies the language.

On the other hand, we can't type-check zig libraries which contain generics. We can only type-check specific uses of those libraries.

```
// This function is typesafe if there exist no odd perfect numbers
// https://en.wikipedia.org/wiki/Perfect_number#Odd_perfect_numbers
fn foo(comptime n: comptime_int, i: usize) usize {
  const j = if (comptime is_odd_perfect_number(n)) "surprise!" else 1;
```

```
    return i + j;
}
```

This means zig also doesn't get the automatic, machine-checked documentation of type constraints that rust benefits from and may face more challenges providing IDE support.

This might harm the zig ecosystem by making it harder to compose various libraries. But julia has a similar model and in practice it has worked very well (eg, eg).

Zig's comptime allows expressing open multiple dispatch as a library.

It should be relatively trivial to implement specialization the same way, which has been a work in progress in rust for years and is critical to many optimizations in julia's math libraries.

Julia chose dynamic typing because it's very difficult to encode the types of various mathematical operations into a general schema (eg fortress struggled with this). Zig's approach of not requiring general schemas but still type-checking individual cases may be an interesting sweet spot.

I used the 2020 CWE Top 25 Most Dangerous Software Weaknesses to get a sense of the relative frequency of different causes of memory unsafety.

(I'm assuming that the zig programmer is using release-safe mode instead of the unfortunately named release-fast mode which disables all runtime safety checks.)

- Out-of-bounds Write (787/1350)
- Out-of-bounds Read (125/1350)
- Improper Restriction of Operations within the Bounds of a Memory Buffer (119/1350)

Both languages primarily use bounds-checked slices and relegate pointer arithmetic to a separate type (`*T` in rust, `[*]T` in zig).

- NULL Pointer Dereference (476/1350)

Both languages require explicit annotations for nulls (`Option<T>` in rust, `?T` in zig) and require code to either handle the null case or safely crash on null (`x.unwrap()` in rust, `x.?` in zig).

Dereferencing/casting a null c pointer is undefined behavior in both languages, but is checked at runtime in zig.

- Integer Overflow or Wraparound (190/1350)

Rust catches overflow in debug and wraps in release. Zig catches overflow in debug/release-safe and leaves behavior undefined in release-fast.

Both languages allow explicitly asking for wraparound (`x.wrapping_add(1)` in rust, `x +% 1` in zig).

- Use After Free (416/1350)

As long as all unsafe code obeys the aliasing and lifetime rules, rust protects completely against UAF.

Zig has little protection. The recently merged GeneralPurposeAllocator avoids reusing memory regions (which prevents freed data from being overwritten) and reusing pages (which means that UAF will eventually result in a page fault). But this comes at the cost of fragmentation and lower performance and it also won't provide protection for child allocators using the GPA as a backing allocator.

Both languages will insert implicit casts between primitive types and pointers whenever it is safe to do so, and require explicit casts otherwise. (With the odd exception that rust will not implicitly upcast numbers).

Both languages support generics which almost entirely avoids the need to cast void pointers.

In rust the Send/Sync traits flag types which are safe to move/share across threads. In the absence of unsafe code it should be impossible to cause data races.

Zig has no comparable protection. It's possible to implement the same logic as Send/Sync in comptime zig, but without the ability to track ownership the rules would have to be much more restrictive.

Rust prevents having multiple mutable references to the same memory region at the same time.

This means that eg iterator invalidation is prevented at compile time, because the borrow checker won't allow mutating a data-structure while an iterator is holding a reference to the data-structure. Similarly for resizing a data-structure while holding a reference to the old allocation. Both examples are easy sources of UAF in zig.

Neither language is able to produce stack traces for stack overflows at the moment (rust, zig)

In the future zig is intended to statically check the maximum stack usage of your program and force recursive code to explicitly allocate space on the heap, so that stack overflows produce a recoverable OutOfMemory error rather than a crash.

This is not an academic problem - I've seen real-world crashes from recursive tree transformations in compilers (eg) and it's often painful to write the same logic without recursion.

Undefined behavior in rust is defined here. It's worth noting that breaking the aliasing rules in unsafe rust can cause undefined behavior but these rules are not yet well-defined. So far this hasn't caused me any problems but it is a little unnerving.

Miri is an interpreter for rusts Mid-level Intermediate Representation which will detect many (but not all) cases of undefined behavior in unsafe rust. It's far too slow to use for the whole materialize test suite but was useful for unit-testing an unsafe module.

Undefined behavior in zig is defined here. This list is probably incomplete given that the core language is still under development.

Zig aspires to insert runtime checks for almost all undefined behavior when compiling in debug mode. So far all the easy cases are handled, which is already a dramatic improvement over c.

Zigs compile-time partial evaluation is done by an IR interpreter - it seems plausible that this could also be used as a miri-like tool in the future.

`@import` takes a path to a file and turns the whole file into a struct. So modules are just structs. And vice-versa - if you have a large struct declaration you can move it into a file to reduce the indentation.

Zig doesn't care at all where you put files on the filesystem.

`@import` is part of the compile-time execution system so things like platform-specific modules and configurable features can be specified in regular code rather than rust's limited set of `#[cfg( ... )]` macros.

Array, struct, enum and union literals can be anonymous - `.{.Constant = 1.0}` is an anonymous union with it's own type, but can be implicitly cast to any union with a `Constant: f64` field because they share the same structure.

In rust my code is littered with `use Expr::*` and I'm careful to avoid name collisions between different enums that I might want to import in the same functions. In zig I just use anonymous literals everywhere and don't worry about it.

Anonymous literals are also nice when using structs to simulate keyword arguments. No need to find and import the correct type:

```
fn do_things(config: struct {
  max_things: usize = 1000, // default value
  flavor: Flavor,
}) void {
  ...
}

do_things(.{.flavor = .Strawberry});
```

There is a pattern that shows up a lot in the materialize codebase:

```
let constant = if let Expr::Constant(constant) = expr { constant } else { panic
```

It's common enough that many types have methods like `expr.unwrap_constant()`.

In zig:

```
const constant = expr.Constant;
```

A similar pattern is:

```
if some_condition {
    if let Expr::Constant(_) = expr {
        ...
    }
}
```

Again, many types get methods like `expr.is_constant()`.

```
if some_condition && expr.is_constant() {
    ...
}
```

In zig:

```
if (some_condition and expr == .Constant) {
    ...
}
```

This works because tagged unions in zig create a separate type for the tag itself and there is an implicit cast from the union to the tag.

Zig integers may be any size (eg `u42` for a 42-bit unsigned integer). Together with the `packed` annotation on structs this makes it really easy to write bit-packing code (eg).

There is no pattern matching, but a `switch` on enums and unions is checked for exhaustiveness at compile time. This covers ~90% of my use of pattern matching.

More complex matches (eg) have to handled by chained `if` statements. This is usually less readable than the equivalent rust and I miss the exhaustiveness checks.

Existing tools for native languages (valgrind, prof etc) work for both languages, modulo name mangling.

Both languages work with gdb and lldb but the experience is mediocre. Rust has builtin support in gdb which improves pretty-printing and allows writing rust expressions, but still has many holes and quirks.

Rust has unstable support for LLVM sanitizers (asan, libfuzzer etc). This is an open issue in zig.

The Rust Language Server is usable. It works well on valid code, but struggles to deal with invalid states during editing. It can also be painfully slow in large codebases - when working on materialize it was often 3-5x slower than just running `cargo check`.

I haven't tried the intellij rust plugin or rust-analyzer.

I haven't yet tried the Zig Language Server but it appears to be fairly complete.

The upcoming incremental zig compiler is intended to be usable as an IDE backend too. I haven't been following the design closely but it seems clear that they've been paying attention to what the rust community has learned over the last half-decade or so.

The zig standard library provides a standard interface for allocators. Everything in the stdlib that allocates memory takes an allocator as an argument. This makes it easy to use custom allocators without having to write your own basic data-structures. While there isn't much of a zig ecosystem yet, it seems likely that future libraries will follow this convention.

Rust provides an implicit global allocator. The standard library and most other libraries use only the global allocator. It's possible to choose the global allocator, but not to use different global allocators in different parts of your program.

While it's possible to write your own allocators in rust, there isn't yet a standard interface for allocators and most of the ecosystem uses only the global allocator. There is work underway to improve the situation.

I've also found that switching from the global allocator to a custom allocator in rust can require a lot of refactoring and produce a lot of boilerplate, as it introduces new lifetimes that spread through every allocated type. When writing compilers in zig I tend to arena allocate everything. In rust I end up using the global allocator and cloning a lot of data instead because it's easier.

In rust, if you store a `String` or `Vec` in eg typed-arena the backing data will still be allocated and freed by the global allocator. It provides the lifetime benefits of an arena allocator but not the performance benefits. Allocators like bumpalo currently require reimplementing stdlib types like `String` and `Vec`. There are plans to fix this.

Until then it's often hard to gain the performance benefits of such specialized allocators eg there is a hot path in materialize' data plane (here) that would be cheaper and simpler if we could just bump allocate `String` and `Vec`.

The rust global allocator panics on allocation failures. Zig allocators return an OutOfMemory error which can be recovered from.

The downside of OutOfMemory errors is that they introduce a *lot* more error paths which can be difficult to test. I mostly choose to just panic anyway, but there are definitely cases where recovery is feasible and useful eg:

- a sql database might want to set a memory limit per client and abort queries that exceed the limit
- a text editor should recover gracefully from trying to open a large file

---

Zig has no syntax for closures. You can create closures manually, but it's verbose. There is a tracking issue open but it doesn't look like there is a consensus yet. The lack of closures limits some kinds of apis eg a direct port of differential dataflow to zig would be unpleasant to use.

Zig has no equivalent to smart pointers. I haven't wanted any yet, but I expect to miss rust's `Rc` eventually. Manually incrementing reference counts is hilariously error-prone.

Zig's error handling model is similar to rust's, but it's errors are an open union type rather than a regular union type like rust's. So:

- The compiler can accurately infer the exact set of errors that can be returned by each function. (In rust it's very tempting to just have a single error enum per library).
- If you handle a subset of possible errors and propagate the rest, the compiler will correctly infer the reduced error set.
- You can use a comptime assert to check that the inferred set of errors is exactly what you expected - neither more nor less.
- It's possible to have name collisions between errors from different libraries!

Zig's errors also carry a trace of each function that they passed through, even if the error changed along the way. This is **not** the same as the stacktrace at the point the error was created - it's tracking the errors path through your error handling code.

To find this path in rust I first have to figure out how to reproduce the crash, record it in rr and then walk backwards from the panic.

Zig's errors can't yet carry any extra information about the error. There is an open issue to improve this but it looks like there are some remaining design problems to be solved.

In the meantime, any extra information has to be passed though a side-channel (eg). This is annoying and error-prone.

And if nothing else, this makes it difficult to implement anyhow's context feature.

Zig prints stacktraces on segfaults!

Generics are implemented as functions that construct types.

```
function List(comptime T: type) type {
    return struct {
        value: T,
        next: ?List(T),

        def len(self: *const List(T)) usize {
          return 1 + if (self.next) |next| next.len() else 0;
        }
    };
}
```

Since modules are just structs, this means you can also emulate ML-style functors.

You can also insert arbitrary logic into that function. Here is a function that implements the struct-of-arrays transformation:

```
fn StructOfArrays(comptime T: type) type {
    // reflect info about the type T
    const t_info = @typeInfo(T);
```

```
    // check that T is a struct
    if (t_info != .Struct) @compileError("StructOfArrays only works on structs!

    // make a new set of fields where each type is an array
    var soa_fields: [t_info.Struct.fields.len]std.builtin.TypeInfo.StructField
    for (t_info.Struct.fields) |t_field, i| {
        var soa_field = t_field;
        soa_field.field_type = []t_field.field_type;
        soa_field.default_value = null;
        soa_fields[i] = soa_field;
    }

    // make a new type with those array fields
    var soa_info = t_info;
    soa_info.Struct.fields = &soa_fields;
    const Inner = @Type(soa_info);

    // return the final type
    return struct {
        inner: Inner,

        const Self = @This();

        /// Fetch the i'th element of self
        fn get(self: *const Self, i: usize) T {
            var t: T = undefined;

            // for each field of t, get the data from the i'th element of the c
            // (this is unrolled at compile time because of the `inline` keywor
            inline for (t_info.Struct.fields) |t_field| {
                @field(t, t_field.name) = @field(self.inner, t_field.name)[i];
            }

            return t;
        }
    };
}
```

While verbose, this is built entirely out of tools that I learned in the first few days of using zig. I think the only way to implement the same example in rust would be with a custom derive but I haven't yet learned how to write procedural macros.

In rust, you can use `#[derive(Ord)]` to create a default implementation of the `Ord` trait for a type. If you don't do this, users of your library **cannot do it themselves** because of the orphan impl rules. If you do do this, users of your library pay for the extra compile time even if they never use it.

The equivalent in zig looks like this:

```
pub fn compare(a: anytype, b: @TypeOf(a)) Ordering {
    const T = @TypeOf(a);
    if (std.meta.trait.hasFn("compare")(T)) {
        // if T has a custom implementation of compare, use that
        return T.compare(a, b);
    } else {
        // otherwise, use reflection to derive a default implementation
        switch (@typeInfo(T)) {
            ...
        }
    }
}
```

(See here for a full example)

This function works on any type, even those from other packages, can be overridden on a type-by-type basis and will only be compiled for functions on which it is actually used. With a little extra work it can also be made extensible by third-party libraries which didn't create the original type.

Zig's compilation is lazy. Only code which is actually reachable needs to typecheck. So if you run `zig test --test-filter the_one_test_i_care_about_right_now` then only the code used for that test needs to typecheck. This makes it much easier to test quick changes and to incrementally refactor code.

Zig has absurdly good support for cross-compiling.

I came across this article while taking a break from repeatedly failing to cross-compile a rust+gtk hello world for the pinephone and had a working zig version within a few hours.

The zig compiler includes sources for it's own standard library and sources for libc for various platforms, so there is no need for a tool like xargo.

On a side note, I was also able to compile the zig+gtk hello world directly on the pinephone, whereas compiling the rust+gtk hello world requires 2.8gb of memory, ~10x as much as the zig version and more than the 2gb available on the phone.

Zig has an experimental build system where the build graph is assembled by zig code. This makes it easy to have various different build tasks and share logic between them. Each task gets tied to a command with some comments which show up in `zig build --help` so it's easy to see what's available.

Rust has a fixed declarative format in `Cargo.toml`. It offers four built-in profiles and doesn't yet allow creating more. It determines the list of targets and which profile to use for each by looking in some predefined directories.

In materialize this doesn't work out - there are more than four profiles needed. Eg one of the test suites is too slow to run without optimizations but it still needs things like overflow checks turned on. It's also often useful to run some target with various different settings. I end up just writing shell scripts that override the profile settings.

`build.zig` also handles tasks like compiling and linking c libraries. In rust, this requires a `build.rs` file which is yet another system to learn.

To be fair, zig doesn't yet have a package manager so it doesn't have to worry about trying to compose build options from various projects. But I expect that specifying the range of options with code rather than a configuration file will still be the simpler choice. Nix and Guix are good prior art here - both make it easy to express arbitrary modifications to dependency graphs by writing code.

Zig has a very streamlined ffi.

Adding a c library requires adding the headers and either the source code or object files to your build.zig (eg) and then importing the header file (eg). Zig can also compile c and c++ code so there isn't any need for additional dependencies. The zig compiler translates the

header files into zig equivalents. Since compilation is lazy there is no need to be able to filter out symbols - anything that isn't used simply won't get compiled.

The rust equivalent is effectively the same process, but it manages to be more complicated by combining a handful of external systems rather than just being built in to the compiler.

In rust, blocks are expressions. Eg `{ a; b }` will return b. In zig blocks it's only possible to return values from a block by using a label `foo: { a; break :foo b; }`. This allows early returns from the block, unlike rust, but I find it harder to read. In rust I sometimes write inner functions to get access to early returns, which is even more verbose.

Overall I prefer the rust approach, but it's a close call.

Zig has various implicit casts that it will make whenever a value of one type is stored in a slot of another type.

For example, there is a cast from anonymous lists to arrays, and from a pointer to an array to a slice.

```
const a: [3]usize = .{0, 1, 2};
const b: []const usize = &a;
```

But zig won't chain these casts, so you can't directly cast an anonymous list to a slice.

```
const c: []const usize = &.{0,1,2};
```

```
./test.zig:62:32: error: expected type '[]const usize', found '*const struct:62
    const c: []const usize = &.{0,1,2};
```

To do this in one line requires the full explicit syntax:

```
&[_]usize{0,1,2};
```

The equivalent is rust is:

```
&[0,1,2];
```

A minor papercut to be sure, but slice literals are everywhere.

Zig has a feature akin to rust's autoborrowing:

```
const Foo = struct {
    ...

    fn bar(self: *Foo) void {
        ...
    }
};

fn quux() void {
    // a stack-allocated Foo
    var foo = Foo { ... };
    // equivalent to Foo.bar(&foo)
    foo.bar();
}
```

This makes it easy to accidentally take a pointer to a stack value, in a way that is not obvious when reading. This is a footgun that is not present in c.

In four months of writing zig, **every** instance of memory unsafety I have detected was either due to:

1. Resizing a collection while holding a pointer to an entry inside it.
2. Using `@fieldParentPtr`.

1 is a mistake I also regularly make in unsafe rust too (eg) - clearly I can't be trusted with pointers :S

2 is unique to zig.

Zig implements dynamic dispatch via a pattern where a struct of function pointers is nested in an outer struct containing closed-over data. The functions can access this data using `@fieldParentPtr`. It's really easy to accidentally move the inner struct to a different location, in which case `@fieldParentPtr` will point at some random location.

This is an acknowledged footgun and there are plans to prevent it.

There is an in-progress incremental debug compiler for zig that aims for sub-second compile times for large projects. Based on progress so far, this is a plausible goal.

Combined with hot code reloading, this would make the feedback loop for zig coding competitive with working in a dynamic, interactive language.

Very anecdotally, because I don't have a good apples-to-apples comparison yet:

Zig in debug mode tends to be around 2x slower than in release mode. I've seen rust in debug mode be up to 10x slower. If this is true in general, a plausible reason for this difference is that many of the 'zero-cost' abstractions that are heavily used in rust (eg iterators) are actually quite expensive without heavy optimization.

Some tests in materialize are so slow in debug mode that I'd end up using release mode instead, for which even an incremental build can be absolutely brutal if I touched something low down in the dependency tree:

```
[nix-shell:~/materialize]$ time cargo build --release --bin sqllogictest
    Finished release [optimized + debuginfo] target(s) in 0.75s

real    0m0.812s
user    0m0.556s
sys     0m0.250s

[nix-shell:~/materialize]$ git diff
diff --git a/src/repr/src/row.rs b/src/repr/src/row.rs
index d71d5198..f492d5ba 100644
--- a/src/repr/src/row.rs
+++ b/src/repr/src/row.rs
@@ -231,7 +231,7 @@ unsafe fn read_datum<'a>(data: &'a [u8], offset: &mut usize
     let tag = read_copy::<Tag>(data, offset);
     match tag {
         Tag::Null => Datum::Null,
-        Tag::False => Datum::False,
+        Tag::False => Datum::True,
```

```
            Tag::True ⇒ Datum::True,
            Tag::Int32 ⇒ {
                let i = read_copy::<i32>(data, offset);

[nix-shell:~/materialize]$ time cargo build --release --bin sqllogictest
   Compiling repr v0.1.0 (/home/jamie/materialize/src/repr)
   Compiling interchange v0.1.0 (/home/jamie/materialize/src/interchange)
   Compiling expr v0.1.0 (/home/jamie/materialize/src/expr)
   Compiling sql-parser v0.1.0 (/home/jamie/materialize/src/sql-parser)
   Compiling pgrepr v0.1.0 (/home/jamie/materialize/src/pgrepr)
   Compiling dataflow-types v0.1.0 (/home/jamie/materialize/src/dataflow-types)
   Compiling sql v0.1.0 (/home/jamie/materialize/src/sql)
   Compiling dataflow v0.1.0 (/home/jamie/materialize/src/dataflow)
   Compiling transform v0.1.0 (/home/jamie/materialize/src/transform)
   Compiling symbiosis v0.1.0 (/home/jamie/materialize/src/symbiosis)
   Compiling coord v0.1.0 (/home/jamie/materialize/src/coord)
   Compiling sqllogictest v0.0.1 (/home/jamie/materialize/src/sqllogictest)
    Building [=======================================================> ] 342/34
    Finished release [optimized + debuginfo] target(s) in 23m 27s

real    23m27.475s
user    136m0.461s
sys     1m55.357s
```

Zig has it's own implementation of standard OS APIs which means that linking libc is completely optional. Among other things, this means that zig can generate very small binaries which might give it an edge for wasm where download/startup times matter a lot.

There isn't yet any way to do coverage-guided fuzzing of zig code. Zig doesn't yet support LLVM sanitizers, including libfuzzer. There is a tracking issue but it faces some architectural challenges.

Some cute examples of using compile-time reflection to reduce boilerplate:

- Iterating over every combination of parse token
- Generating AST visitors
- Reversing a finite function
- Generating arbitrary types from fuzzer-provided bytes

Defer runs at the end of the lexical scope it was declared in, rather than at the end of the function like go.

So far I haven't had any difficulty using defer to clean up resources, but on the other hand most of the code I've written leans heavily on arena allocation and I also haven't put much

effort into testing error paths yet. I don't expect to have much of an opinion either way until I've written a lot more code and properly stress tested some of it.

I suspect that defer will be the easy part, and the hard part will be making sure that every try has matching errdefers. There's a bit of an asymmetry between how easy it is to propagate errors and how hard it is to handle and test resource cleanup in all those propagation paths.

Main points so far:

- Zig has much lower cognitive overhead than rust and I'm more able to directly express things I care about.
- Zig seems much more likely than rust to achieve very fast feedback loops (eg sub-second hot code reloading for large projects).
- Zig's compile-time reflection can directly implement many features that I've had to use adhoc code generation for in the past (eg data-structures that vary their layout depending on the element type) and features that in rust require extending the compiler (eg generic functions with specialized implementations for specific types).
- Zig currently has an edge in terms of custom allocators and handling out-of-memory errors. Rust has plans to improve but it remains to be seen whether the new apis will percolate through the whole ecosystem or whether they'll mostly be seen in the much smaller `no_std` ecosystem.
- Zig has no mechanism for preventing UAF escalating into vulnerabilities. The new GPA can detect some forms of UAF but the fragmentation overhead is too high for production use.
- Rust is better able to protect against some other classes of bugs outside of UAF, especially data races, but also eg iterator invalidation.
- Rust makes it much harder to leak memory/resources, especially when dealing with error paths.

Most of the places where zig is simpler than rust do not relate to the lifetime system, and I'm not yet convinced by arguments that all of rust's other features are necessary to support the lifetime system (eg zig does not have traits, but can still easily implement equivalents to rust's `Drop` and `Deref` traits). It may be possible to implement a lifetime system on top of zig. But this would be a big change to the language late in the game and much of the standard library would probably have to change to accommodate it.

A more promising option is supporting something like nanoprocesses. This wouldn't prevent UAF but would make it harder to exploit by forcing the attacker to break out of several sandboxes before gaining access to useful functions. This is something I might try working on once the self-hosted compiler is complete.

jamie@scattered-thoughts.net
scattered-thoughts.net/rss.xml