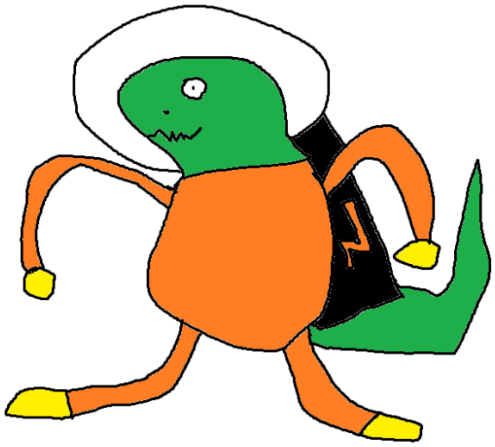

[1]

Writing a grep like program in Zig

Seminar Programming Languages

Tobias Schmitz

2024-01-07

Contents

1. Introduction

The objective of this seminar was getting to know a new programming language by writing a simple grep [2] like program.

2. Language

Zig is a general-purpose compiled systems programming language [3]. It was initially developed by Andrew Kelley and released in 2016 [4]. Today development is funded by the Zig software foundation (ZSF), which is a nonprofit (`501(c)(3)`) corporation stationed in New York [5].

Zig is placed as a successor to C, it is an intentionally small and simple language, with its whole syntax fitting in a 500 line PEG grammar file [6]. If focuses on readability and maintainability restricting the control flow only to language keywords and function calls [7].

2.1. Toolchain

Installation was as simple as downloading the release tar archive from the downloads section of the Zig website [8], extracting the toolchain, and symlinking the binary onto a `$PATH`. There is also a community project named `zigup` [9] which is allows installing and managing multiple versions of the Zig compiler.

The compiler is a single executable named `zig`, it includes a build system which can be configured in a `build.zig`, which is written in Zig [10]. The toolchain itself is also a C/C++ compiler which allows Zig code to directly interoperate with existing C/C++ code. [3]

To start a new project inside an existing directory running `zig init-exe` will generate the main source file `src/main.zig` and the build configuration `build.zig`. The program can then be built and executed by running `zig build run`. [11]

The Zig community also provides a language server named `zls` [12], which worked right away after setting it up in `neovim` [13]. There were some issues with type inference, and completion of member functions of generic types. In some cases `zls` would report no errors when the Zig compiler would.

2.2. Integers

Zig has concrete integer types with a predetermined size regardless of the target platform. Commonly used types are:

- unsigned: `u8`, `u16`, `u32`, `u64`
- signed: `i8`, `i16`, `i32`, `i64`.

But it also allows defining both signed and unsigned, arbitrary sized integers like `i27`, or `u3`, up to a maximum bit-width of `65535` [14].

2.3. Arrays and slices

Zig arrays [15] have a size known at compile time and are stack allocated, unless specified otherwise:

```
1  const explicit_length = [5]u32{ 0, 1, 2, 3, 4 };
2  const inferred_length = [_]u32{ 5, 6, 7, 8, 9 };
```

Arrays can be sliced using the index operator with an end-exclusive or half-open range, returning a slice [16] to the referenced array. By default the slice is a fat pointer which is composed of a pointer that points to the memory address of the slice inside the array, and the length of the slice.

```zig
const array = [_]u32{ 0, 1, 2, 3, 4 };
const slice = array[1..3];
std.debug.print("{*}\n{d}\n", .{ slice, slice.* });
```

The code prints the fat pointer itself, and the dereferenced values of the array it points to:

```
[2]u32@2036fc
{ 1, 2 }
```

For better interoperability with C Zig also allows sentinel terminated pointers [17] or slices. Most commonly this is used for null-terminated strings:

```zig
const string: *const [32:0]u8 = "this is a null terminated string";
const fat_pointer_slice: []const u8 = string[10..];
const null_term_slice: [:0]const u8 = string[10..];
const null_term_ptr: [*:0]const u8 = string[10..];

std.debug.print("size: {}, '{s}'\n", .{@sizeOf([]const u8), fat_pointer_slice});
std.debug.print("size: {}, '{s}'\n", .{@sizeOf([:0]const u8), null_term_slice});
std.debug.print("size: {}, '{s}'\n", .{@sizeOf([*:0]const u8), null_term_ptr});
```

The code prints the size of the pointer and the string it references:

```
size: 16, 'null terminated string'
size: 16, 'null terminated string'
size: 8, 'null terminated string'
```

All three slice/pointer types reference the same data, but in a different way. Variant 1 uses the fat pointer approach described above. Variant 2 uses the same approach but also upholds the constraint that the end of the slice is terminated by a null-byte sentinel. Variant 3 only stores a memory address and relies upon the null-byte sentinel to compute the length of the referenced data when needed.
On 64-bit target platforms the first and the second slice type have a size of 16 bytes, 8 bytes for the pointer and 8 additional bytes for the length. The sentinel terminated pointer only has a size of 8 bytes, since it does not store an additional length field.

A limitation of sentinel terminated slices or pointers is that they cannot reference arbitrary parts of an array. Trying to do so fails with the following message:

```
src/main.zig:10:62: error: expected type '[:0]const u8', found '*const [13]u8'
    const null_terminated_string_slice: [:0]const u8 = string[10..23];
                                                        ~~~~~~^~~~~~~~
src/main.zig:10:62: note: destination pointer requires '0' sentinel
```

2.4. Tagged unions

In Zig tagged unions are very similar to Rust enums [18]. The tag can be either an existing enum or inferred from the union definition itself. If an existing enum is used as a tag, the compiler enforces that every variant that the enum defines is present in the union declaration. Tagged unions can be coerced to their enum tag, and an enum tag can be coerced to a tagged union when it is known at `comptime` and the union variant type has only one possible value such as `void`. [19]

```
const TokenType = enum {
    Ident,
    IntLiteral,
    Dot,
};
const Token = union(TokenType) {
    Ident: []const u8,
    IntLiteral: u64,
    Dot,
};

const ident_token = Token { .Ident = "abc" };
const token_type: TokenType = ident_token;
// `Token.Dot` has only one value
const dot_token: Token = TokenType.Dot;
```

2.5. Type inference

The type of Zig variables is inferred using only directly assigned values. A type can optionally be specified, and is required in some cases. For example when an integer literal is assigned to a mutable variable, its exact type must be specified. When the type of a `struct` is known, such as when passing it to a function, its name can be omitted and an anonymous literal can be used:

```
const Foo = struct {
    a: i32,
    b: bool = false,
};
fn doSomething(value: Foo) void { ... }

doSomething(.{ .a = 21 });
```

The same is also true for an `enum` and a tagged `union`. When the type is known, the name of the `enum` can be omitted and only the variant needs to written out:

```
const Bar = enum {
    One,
    Two,
};
const BAR_TWO: Bar = .Two;

const Value = union(enum) {
    Int: u32,
    Float: f32,
    Bool: bool,
};
const INT_VAL: Value = .{ .Int = 324 };
```

2.6. Control flow

Zig has special control flow constructs for dealing with union and optional values. Optional values can be passed into an if statement and if a non-null value is contained, it can be accessed inside the if statement's body:

```
const optional_int: ?u32 = 86;
if (optional_int) |int| {
    std.debug.print("Contains int {}\n", .{int});
}
```

To unwrap optional values more conveniently zig provides the `orelse` operator which allows specifying a default value, and the `.?` operator which forces a value to be unwrapped and will crash if it is `null`.

```
const optional_token: ?Token = parser.next();
const token = optional_token orelse unreachable;
// syntax sugar for the above
const token = optional_token.?;
```

Switch statements can be used to extract the values of tagged unions in a similar way:

```
switch (token) {
    .Ident => |ident| {
        std.debug.print("Identifier: '{}'", .{ident});
    }
    .IntLiteral => |int| {
        std.debug.print("Integer literal: '{}'", .{int});
    }
    .Dot => {},
}
```

2.7. Error handling

In Zig there are no exceptions and errors are treated as values. If a function is fallible it returns an error union, the error set of that union can either be inferred or explicitly defined.

```
// inferred error set
pub fn main() !void {
    const num = try std.fmt.parseInt(u32, "324", 10);
    //          ^ unwraps or returns the error
    std.debug.print("{d}\n", .{num});
}

// named error set
const IntError = error{
    IsNull,
    Invalid,
};
fn checkNull(num: usize) IntError!void {
    if (num == 0) {
        return error.IsNull;
    }
}
```

Like optional values, error unions can be inspected using an if statement. The only difference is that the else clause now also grants access to the error value:

```
fn fallibleFunction() !f64 { ... }

if (fallibleFunction()) |val| {
    std.debug.print("Found value: {}\n", .{val});
} else |err| {
    std.debug.print("Found error: {}\n", .{err});
}
```

And likewise the `catch` operator for error unions corresponds to the `orelse` operator of optionals.

```
const DEFAULT_PATH = "$HOME/.config/zig-grep";
const path = readEnvPath() catch DEFAULT_PATH;
```

Additionally the `catch` operator allows capturing the error value.
Similar to Rust [20] Zig has a `try` operator that either returns the error of an error union from the current function or unwraps the value:

```
const file = openFile() catch |err| return err;
// syntax sugar for the above
const file = try openFile();
```

2.8. Defer

There are no constructors or destructors in Zig, so unlike C++ where the RAII [21] model is often used to make objects manage their resources automatically, resources have to be managed manually. A commonly used pattern to manage resources is for an object to define `init` and a `deinit` procedures, which have to be called manually. In that case the `init` procedure is a static member function on the type that returns an instance of the type, and the `deinit` procedure is a method of the object.

To make this more ergonomic Zig provides `defer` statements, which allow running cleanup code when a value goes out of scope. If multiple `defer` statements are defined, they are run in reverse declaration order. This allows the deinitialization code to be directly below the initialization: [22]

```
fn fallibeFunction() ![]const u8 {
    var foo = try std.fs.cwd().openFile("foo.txt", .{});
    defer foo.close();
    var bar = try std.fs.cwd().openFile("bar.txt", .{});
    defer bar.close();

    // the defer statements will be executed in this order:
    // 1. bar.close();
    // 2. foo.close();
}
```

The `errdefer` statement runs code **only** when an error is returned from the scope, this can be useful when dealing with a multi step initialization process that can fail, and intermediate resources need to be cleaned up: [23]

```
fn openAndPrepareFile() !File {
    var file = try std.fs.cwd().openFile("foo.txt", .{});
    errdefer file.close();
    try file.seekBy(8);
    return file;
}
```

If the function succeeds the file is returned from it, so it should not be closed. If it fails while seeking and returns an error, the `errdefer` statement is executed and the file is closed as to not leak any resources.

2.9. Memory management

Zig does not include a garbage collector and uses a very explicit manual memory management strategy. Memory is manually allocated and deallocated via `Allocator`s that are choosen and instantiated by the user. Data structures or functions which might allocate require an allocator to be passed explicitly. The standard library includes a range of allocators fit for different use cases, ranging from general purpose bucket allocators, bump- or arena allocators, to fixed buffer allocators.
Memory allocation may fail, and out of memory errors must be handled. Memory deallocation must always succeed. Like other resources, data structures that allocate commonly provide an `init` and `deinit` procedure which can be used in combination with a `defer` statement to make sure allocated memory is freed.
In debug mode the `GeneralPurposeAllocator` keeps track of allocations and detects memory leaks and double frees [24].

2.10. Comptime

Zig provides a powerful compile time evaluation mechanism to avoid using macros or code generation. Contrary to C++ or Rust where functions have to be declared `constexpr` [25] or `const` [26] in order to be called at compile time, in Zig everything that can be evaluated at `comptime` just is. A function called from a `comptime` context will either yield an error explaining what part of it is not able to be evaluated at `comptime` or evaluate the value during compilation. A `comptime` context can be a constant defined at the global scope, or a `comptime` block. [27]

This can be used to do expensive calculations, generate lookup tables, or uphold constraints using assertions, all at compile time:

```
const FIB_8 = fibonacci(8);
comptime {
    // won't compile
    std.debug.assert(fibonacci(3) == 1);
}

fn fibonacci(n: usize) u64 {
    if (n == 0 or n == 1) {
        return 1;
    }
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

This means, if the main function is able to be evaluated at compile time, and it is called from a `comptime` context, the Zig compiler acts as an interpreter and the program is executed during compilation. Granted since comptime code can not perform I/O such a program is quite limited.

Arguments to functions can be declared as `comptime` which requires them to be known during compilation. This is often used for types passed to function in the same way other languages handle generics. Considering the following Kotlin [28] class:

```
class Container<T>(
    var items: ArrayList<T>,
)
```

An equivalent Zig `struct` would be defined as a function taking a `comptime` type as an argument that returns another type.

```
fn Container(comptime T: type) type {
    return struct {
        items: ArrayList(T),
    }
}
```

Note that ArrayList is another such function defined in the standard library.

2.11. SIMD

In addition to the automatic vectorization of code that the LLVM [29] optimizer does, Zig also provides a way to explicitly define vector operations, using the `@Vector` intrinsic, that will compile down to target specific SIMD operations: [30]

```
const a = @Vector(4, i32){ 1, 2, 3, 4 };
const b = @Vector(4, i32){ 5, 6, 7, 8 };
const c = a + b;
```

2.12. Ecosystem

Compared to C++, Java, or Rust the standard library of Zig is quite minimal. Neither the Zig language itself, nor the standard library directly define a string datatype. String literals are represented as byte slices (`[]const u8`), which allows using the whole range of `std.mem.*` functions to operate on them.

With Zig being a young language, the eco system in general is still a little immature. To date there is no regex library written in zig that has feature parity with established regex engines. Zig `0.11.0` does not include support for `async` functions [31].

3. Development process

Since the scope of the program was predetermined, I mainly focused on performance.

3.1. Directory walking

The Zig standard library provides `IterableDir`, an iterator for traversing a directory in a depth first manner, but unfortunately that approach does not allow filtering of searched directories. To overcome that limitation I mostly copied the standard library function for walking directories and modified it slightly to allow filtering out hidden directories.

3.2. Linking and using the regex engine

Since there are no usable regex engines written in Zig I had to find a library written in another language. I decided on using the Rust regex library (`rure`) [32] through its C API, because it is a standalone project, easy to build [33], and reasonably fast [34].
To include a C library, some modification inside the `build.zig` configuration file are needed:

```
// link all the other stuff needed
exe.linkLibC();
exe.linkSystemLibrary("util");
exe.linkSystemLibrary("dl");
exe.linkSystemLibrary("gcc_s");
exe.linkSystemLibrary("m");
exe.linkSystemLibrary("rt");
exe.linkSystemLibrary("util");
// link rure itself
exe.addIncludePath(LazyPath.relative("rure/regex-capi/include"));
exe.addLibraryPath(LazyPath.relative("rure/target/release"));
exe.linkSystemLibrary2("rure", .{
    .needed = true,
    .preferred_link_mode = .Static,
});
```

This links the `rure` crate statically into the final binary.

The dependencies are taken straight from the `rure` compile script:

```
# N.B. Add `--release` flag to `cargo build` to make the example run faster.
cargo build --manifest-path ../Cargo.toml
gcc -O3 -DDEBUG -o iter iter.c -ansi -Wall -I../include -L../../target/debug -
lrure
# If you're using librure.a, then you'll need to link other stuff:
# -lutil -ldl -lpthread -lgcc_s -lc -lm -lrt -lutil -lrure
```

When linking C libraries, Zig is not able to include debug symbols, so crash messages that would normally be informative, only show memory addresses:

```
thread 20843 panic: index out of bounds: index 14958, len 14948
Unwind error at address `:0x2ebaef` (error.InvalidDebugInfo), trace may be
incomplete
```

This is a known issue [35].

The C functions can then be imported using the `@cImport` intrinsic:

```
const c = @cImport({
    @cInclude("rure.h");
});
```

And the C definitions can be accessed using the returned object.

```
var match: c.rure_match = undefined;
const found = c.rure_find(ctx.regex, @ptrCast(text), text.len, pos, &match);
```

3.3. Single threaded optimization

The program was profiled in an end to end manner using `hyperfine` [36].

3.3.1. Line by line searching

To keep it simple, the first implementation reads the whole file into a single buffer and runs a pre-compiled regex search on every line. Regex pattern matching is done using a regex iterator from the `rure` crate.

3.3.2. Whole text searching

After some investigation it turned out that initializing the regex iterator provided by the `rure` crate had more overhead than expected, and running the regex search on the whole text instead of every line would improve performance significantly. Following this change, I found out that the `rure` library also provided a function that allowed searching from a specified start index inside the passed text slice. Using this function avoided allocating the iterator in the first place.

3.3.3. Line by line searching with fixed size buffer

Since one of the tests was to search an 8Gb large text file, the input would need to be split up into smaller chunks as to avoid running out of memory. This is done using a fixed size buffer which only loads part of the file, searching that buffer up to the last fully included line, then moving the unsearched parts including possibly relevant context lines to the start of the buffer, and eventually refilling the buffer with remaining data to search. Since lines need to be iterated anyway to calculate line numbers, and the implementation was now using the function that searches the text directly without an iterator, I decided to once again search each line individually, instead of the whole text.

3.3.4. Whole text searching with fixed size buffer

After further investigation I discovered that the overhead of searching each line did not just come from the `rure` iterator, but that special regex patterns introduced large overhead when starting the search. One example was the word character pattern `\w`, which has to respect possibly multi-byte unicode characters. Since the `rure` library uses a finite automata (state machine), matching multiple word characters results in a large number of states [37]. This state machine, although only compiled once, needs to be initialized in memory every time a search is started. Disabling unicode support during the regex pattern compilation significantly improves performance. With these findings, the regex pattern matching was once again adjusted to be run on the whole text buffer, to restore previously achieved performance.

One additional bug that I only tackled at this stage was to prevent regex matches that spanned multiple lines. If a match is found that spans multiple lines an additional search is run only on the first matched line, if it succeeds too, only this match is highlighted and printed, otherwise it is a false positive.

3.4. Parallelization

At this point most easy wins in single threaded optimization were off the table, so the next major performance improvements would come from using multiple threads. The most time consuming sections of the program are accessing the file system, and searching the text.

Parallelization of text searching was implemented using a thread pool of workers that would receive file paths through an atomically synchronized, ring-buffer message queue (`AtomicQueue` in `src/atomic.zig`). The message queue synchronization is implemented using a mutex for exclusive access and use a futex [38] as an event system to notify other workers when the state has changed. Workers wait for a new message from the queue and receive either a path to search or a stop signal:

```
while (true) {
    var msg = ctx.queue.get();
    switch (msg) {
        .Some => |path| {
            defer ctx.allocator.free(path.abs);
            try searchFile(&ctx, text_buf, &line_buf, &path);
        },
        .Stop => break,
    }
}
```

The directory walking remains mostly the same apart from searching files ad hoc, they were now sent through the message queue.

Since there are now multiple threads writing to `stdout` their output has to be synchronized so that lines from one file would not be interspersed with other ones.
There are two obvious solutions to this problem. One is to use a dynamically growing allocated buffer which stores the entire output of a searched file and then write the entire buffer in a synchronized way when the file is fully searched. This would avoid blocking other threads, but could cause the program to run out of memory if large portions of big files would match a search pattern.
The other solution is to just block output of all other threads once a match has been found in a file and then write all lines directly to `stdout`. This would avoid running out of memory, but could in worst case scenarios cause basically single threaded performance.
The final implementation uses a hybrid of the two, each thread has a fixed size output buffer which can be written to without any locking (`SinkBuf` in `src/atomic.zig`). Once the buffer is full, access to `stdout` is locked using the underlying thread safe writer (`Sink` in `src/atomic.zig`) and the thread is free to write to it until the file is fully searched. While `stdout` is locked, other workers can still make progress and access their thread-local output buffers.

With only text searching parallelized the search workers were consuming messages from the queue faster than paths could be added, so the goal was to speed up walking the file system with multiple threads.
This was heavily influenced by the Rust `ignore` crate [39] which is also used in `ripgrep` [40]. A thread pool of walkers is used to search multiple directories simultaneously in a depth first manner to reduce memory consumption.
The core data structure used is an atomically synchronized, priority stack (`AtomicStack` in `src/atomic.zig`). A walker tries to pop off a directory of a shared atomically synchronized stack, by blocking until one is available. Once it receives a directory it iterates through the remaining entries, enqueuing any files encountered. If it encounters a subdirectory, the parent directory is pushed back onto the stack and the subdirectory is walked. The stack keeps track of the number of waiting threads

and once all walkers are waiting for a new message, all directories have been walked completely and the thread pool is stopped:

```
self.alive_workers -= 1;
if (self.alive_workers == 0) {
    self.state.store(@intFromEnum(State.Stop), std.atomic.Ordering.SeqCst);
    Futex.wake(&self.state, std.math.maxInt(u32));
    return .Stop;
}
```

3.5. Command line argument parsing

Argument parsing makes use of tagged unions and `comptime`.

There are two different types of arguments: flags and values, both of these are defined as enums. While flags are just boolean toggles, values require for example a number, to be specified after them. `UserArgKind` is a tagged union that contains either one or the other:

```
const UserArgKind = union(enum) {
    value: UserArgValue,
    flag: UserArgFlag,
};

const UserArgValue = enum(u8) {
    Context,
    AfterContext,
    BeforeContext,
};
const UserArgFlag = enum(u8) {
    Hidden,
    FollowLinks,
    Color,
    ...
};
```

All user arguments are defined in an array, including their long form, an optional short form, a description and their union representation:

```
const USER_ARGS = [_]UserArg{
    .{
        .short = 'A',
        .long = "after-context",
        .kind = .{ .value = .AfterContext },
        .help = "prints the given number of following lines for each match",
    },
    ...
    .{
        .short = null,
        .long = "help",
        .kind = .{ .flag = .Help },
        .help = "print this message",
    },
    ...
};
```

When parsing command line arguments this can be used to exhaustively match all possible valid inputs using a switch statement. When adding a new enum variant the compiler enforces it is handled in all switch statements that match the modified enum. This is the simplified switch statements that handles all arguments:

```
switch (user_arg.kind) {
    .value => |kind| {
        switch (kind) {
            .Context => {
                opts.after_context = num;
                opts.before_context = num;
            },
            ...
        }
    },
    .flag => |kind| {
        switch (kind) {
            .Hidden => opts.hidden = true,
            ...
        }
    },
}
```

The help message is generated at `comptime`, using the list of possible arguments.
Instead of a general purpose allocator a fixed buffer allocator had to be used, but otherwise the code could be written without taking any precautions.

3.6. Compiler bug

With Zig `0.11.0` I encountered a bug in the compiler which would affect command line argument parsing. In debug mode arguments were parsed as expected, but in release mode the `--ignore-case` flag would be parsed as the `--hidden` flag. All flags are defined as an `enum`:

```
const UserArgFlag = enum {
    Hidden,
    FollowLinks,
    Color,
    NoHeading,
    IgnoreCase,
    Debug,
    NoUnicode,
    Help,
};
```

The issue was fixed by specifying a concrete tag type to represent the enum instead of letting the compiler infer the type:

```
@@ -27,12 +27,12 @@ const UserArgKind = union(enum) {
     flag: UserArgFlag,
 };

-const UserArgValue = enum {
+const UserArgValue = enum(u8) {
     Context,
     AfterContext,
     BeforeContext,
 };
-const UserArgFlag = enum {
+const UserArgFlag = enum(u8) {
     Hidden,
     FollowLinks,
     Color,
```

At the time I discovered the bug, it was already fixed on the Zig `master` branch.
I was not able to find a github issue or a pull request related to this bug, but my best guess is that the enum was somehow truncated to two bits, which would strip the topmost bit of the `IgnoreCase` variant represented as `0b100`, resulting in `0b00` which corresponds to `Hidden`.

4. Conclusion

4.1. Program

Since the Zig compiler as of version `0.11.0` is not able to genrate debug symbols when a C library is linked, I was not able to profile the program more thoroughly. I would have liked to look at a `flamegraph` [41] and more detailed timing information to optimize the application.

The program also currently synchronizes output for entire files, even if the `--no-heading` option is enabled. It should be faster to only synchronize output for lines when that is the case. Especially when a large file might be blocking other threads from progressing to the next file, because it has filled its output buffer and prevents them from writing to `stdout`.

Apart from that learning Zig was an interesting journey, which definitely was a breath of fresh air for someone who mostly writes Rust code. I was able to write a program that is only 2 times as slow as `ripgrep` [40] when run on test cases provided by the task Section 6.

4.2. Zig

While having several constructs that make it easier to write memory safe code than C, like optional types, `defer` statements, or a slice type with a length field, Zig is still an unsafe language regarding memory management. Compared to managed languages with garbage collectors or Rust that has hard rules in place to avoid double frees, data races, and to some degree memory leaks, a program written in Zig still places a burden on the programmer to avoid memory related bugs.

But this is done for a reason, Zig allows competent programmers to write high performance code while taking full control of the system. It does so while being more ergonomic than C and being less constraining than Rust.

5. Bibliography

[1] Andrew Kelley, *zigfast.png*. 2020. [Online]. Available: https://github.com/ziglang/gotta-go-fast/commit/a117b3a120a41cb9f0c6ba2fd4a509360ef69185#diff-5e421635b77da50cca212633074ca9e36df775d1ac3cecd3a99b1b33aed45519

[2] "GNU Grep". Jan. 03, 2024. [Online]. Available: https://www.gnu.org/software/grep/manual/grep.html

[3] "Zig". [Online]. Available: https://ziglang.org/

[4] Andrew Kelley, "Introduction to the Zig Programming Language". Feb. 08, 2016. [Online]. Available: https://andrewkelley.me/post/intro-to-zig.html

[5] "Sponsoring the Zig Software Foundation". [Online]. Available: https://ziglang.org/zsf/

[6] "Grammar". Jan. 05, 2024. [Online]. Available: https://ziglang.org/documentation/master/#Grammar

[7] "In-depth Overview", Zig. [Online]. Available: https://ziglang.org/learn/overview/

[8] "Releases", Zig. [Online]. Available: https://ziglang.org/download/

[9] "zigup". Jan. 03, 2024. [Online]. Available: https://github.com/marler8997/zigup

[10] "Zig Build System", Zig. [Online]. Available: https://ziglang.org/learn/build-system/

[11] "Getting Started", Zig. [Online]. Available: https://ziglang.org/learn/getting-started/

[12] "zls". Jan. 03, 2024. [Online]. Available: https://github.com/zigtools/zls

[13] "neovim". [Online]. Available: https://neovim.io/

[14] "Integers". Jan. 03, 2024. [Online]. Available: https://ziglang.org/documentation/0.11.0/#Integers

[15] "Arrays". Jan. 03, 2024. [Online]. Available: https://ziglang.org/documentation/0.11.0/#Arrays

[16] "Slices". Jan. 03, 2024. [Online]. Available: https://ziglang.org/documentation/0.11.0/#Slices

[17] "Pointers". Jan. 03, 2024. [Online]. Available: https://ziglang.org/documentation/0.11.0/#Pointers

[18] "Enums and Pattern Matching". [Online]. Available: https://doc.rust-lang.org/book/ch06-00-enums.html

[19] "Tagged union". Jan. 03, 2024. [Online]. Available: https://ziglang.org/documentation/0.11.0/#Tagged-union

[20] "A Shortcut for Propagating Errors: the ? Operator". Jan. 04, 2024. [Online]. Available: https://doc.rust-lang.org/book/ch09-02-recoverable-errors-with-result.html#a-shortcut-for-propagating-errors-the--operator

[21] "RAII". Jan. 04, 2024. [Online]. Available: https://en.cppreference.com/w/cpp/language/raii

[22] "defer". Jan. 04, 2024. [Online]. Available: https://ziglang.org/documentation/0.11.0/#defer

[23] "errdefer". Jan. 04, 2024. [Online]. Available: https://ziglang.org/documentation/0.11.0/#errdefer

[24] "GeneralPurposeAllocator source code". Jan. 04, 2024. [Online]. Available: https://ziglang.org/documentation/0.11.0/std/src/std/heap/general_purpose_allocator.zig.html

[25] "constexpr specifier". Jan. 04, 2024. [Online]. Available: https://en.cppreference.com/w/cpp/language/constexpr

[26] “Constant evaluation”. Jan. 04, 2024. [Online]. Available: https://doc.rust-lang.org/reference/const_eval.html

[27] “comptime”. Jan. 04, 2024. [Online]. Available: https://ziglang.org/documentation/0.11.0/#comptime

[28] “Kotlin”. [Online]. Available: https://kotlinlang.org/

[29] “LLVM”. [Online]. Available: https://llvm.org/

[30] “Vectors”. Jan. 04, 2024. [Online]. Available: https://ziglang.org/documentation/0.11.0/#Vectors

[31] Andrew Kelley, “The Upcoming Release Postponed Two More Weeks and Lacks Async Functions”. Jul. 19, 2023. [Online]. Available: https://ziglang.org/news/0.11.0-postponed-again/

[32] “rust-lang/regex”. Jan. 03, 2024. [Online]. Available: https://github.com/rust-lang/regex

[33] “Building and Running a Cargo Project”, The Rust Programming Language. [Online]. Available: https://doc.rust-lang.org/book/ch01-03-hello-cargo.html#building-and-running-a-cargo-project

[34] Andrew Gallant (BurntSushi), “rebar”. Jan. 03, 2024. [Online]. Available: https://github.com/BurntSushi/rebar

[35] “Invalid debug info when linking to system library.”. [Online]. Available: https://github.com/ziglang/zig/issues/12046

[36] “hyperfine”. Jan. 05, 2024. [Online]. Available: https://github.com/sharkdp/hyperfine

[37] “Simple regex like \w256 take tens of milliseconds to compile”. [Online]. Available: https://github.com/rust-lang/regex/issues/1095

[38] “futex - fast userspace locking”. Jan. 05, 2024. [Online]. Available: https://www.man7.org/linux/man-pages/man7/futex.7.html

[39] “ignore”. Jan. 05, 2024. [Online]. Available: https://github.com/BurntSushi/ripgrep/tree/master/crates/ignore

[40] “ripgrep”. Jan. 03, 2024. [Online]. Available: https://github.com/BurntSushi/ripgrep/

[41] “Flame Graphs”. [Online]. Available: https://brendangregg.com/flamegraphs.html

6. Benchmark results

The raw benchmark output can be found in the `bench` directory.

Benchmarks were run using `hyperfine` and a modified version of the `test.py` script.
The exact command used was:

```
python test/test.py --ripgrep -v --bench --fail-fast -d test/data ./zig-out/bin/zig-grep
```

6.1. Result from a thinkpad with an Ryzen 7 5800u and 16Gb ram

	ripgrep [ms]			searcher [ms]			
name	min	avg	max	min	avg	max	%
literal_linux	149	162	211	341	346	360	213
literal_linux_hidden	174	179	191	405	417	430	231
linux_literal_ignore_case	172	176	185	391	400	405	227
linux_pattern_prefix	164	168	179	381	394	398	234
linux_pattern_prefix_with_context	171	174	181	383	394	399	225
linux_pattern_prefix_ignore_case	179	186	192	431	446	457	239
linux_pattern_suffix	172	178	184	528	537	544	301
linux_pattern_suffix_with_context	194	198	206	547	562	569	282
linux_pattern_suffix_ignore_case	183	191	195	573	588	594	307
linux_word	163	170	177	375	389	397	229
linux_word_with_heading	164	168	179	378	387	392	230
linux_word_ignore_case	178	181	188	618	636	646	350
linux_no_literal	379	385	386	596	609	618	158
linux_no_literal_ignore_case	373	386	397	599	614	657	159
linux_alternatives	173	177	184	393	402	407	226
linux_alternatives_with_heading	172	177	184	393	400	407	226
linux_alternatives_ignore_case	200	203	206	420	427	432	210
subtitles_literal	7614	10099	12845	9681	9931	10534	98
subtitles_literal_ignore_case	8288	9027	9977	9969	10276	11202	113
subtitles_alternatives	7959	9245	10024	10435	10646	11088	115
subtitles_alternatives_ignore_case	8973	9735	10353	12594	12921	13717	132
subtitles_surrounding_words	8186	9026	9742	9737	9910	10160	109
subtitles_surrounding_words_ignore_case	7959	8795	9736	10451	10740	11166	122
subtitles_no_literal	24069	24889	25574	27419	27628	27981	111
subtitles_no_literal_ignore_case	24592	25243	26059	27501	27692	28037	109

Average runtime compared to ripgrep: 198%.

6.2. Result from a desktop with an Ryzen 5 5600x and 32Gb ram

	ripgrep [ms]			searcher [ms]			
name	min	avg	max	min	avg	max	%
literal_linux	101	103	105	255	258	261	250
literal_linux_hidden	109	111	120	302	378	560	339
linux_literal_ignore_case	107	111	125	285	308	334	276
linux_pattern_prefix	108	123	141	277	315	396	255
linux_pattern_prefix_with_context	116	121	136	279	290	325	239
linux_pattern_prefix_ignore_case	236	242	258	316	322	330	132
linux_pattern_suffix	116	118	121	394	410	450	345
linux_pattern_suffix_with_context	129	132	134	413	425	433	322
linux_pattern_suffix_ignore_case	123	125	130	430	436	445	348
linux_word	109	112	116	271	275	280	245
linux_word_with_heading	110	112	115	274	284	298	251
linux_word_ignore_case	118	121	123	476	484	490	399
linux_no_literal	367	373	389	451	460	473	123
linux_no_literal_ignore_case	371	377	384	451	459	473	121
linux_alternatives	113	116	123	291	302	315	259
linux_alternatives_with_heading	113	116	121	285	289	295	248
linux_alternatives_ignore_case	174	176	178	305	311	320	176
subtitles_literal	1695	2051	4688	7495	7584	7656	369
subtitles_literal_ignore_case	2286	2307	2331	7976	8087	8173	350
subtitles_alternatives	1973	1995	2037	7939	8130	8238	407
subtitles_alternatives_ignore_case	3908	3927	3968	9434	9579	9802	243
subtitles_surrounding_words	1688	1710	1728	7411	7565	7701	442
subtitles_surrounding_words_ignore_case	2078	2091	2109	7936	8098	8238	387
subtitles_no_literal	18931	18949	18976	23839	24089	24387	127
subtitles_no_literal_ignore_case	18905	18923	18946	23863	24213	24411	127

Average runtime compared to ripgrep: 271%.