Seminar Programmiersprachen im Verlgeich - Zig

Tobias Schmitz

Contents

1. Introduction

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

2. Language

2.1. About

Zig is a general-purpose compiled systems programming language. It was initially developed by Andrew Kelley and released in 2015/2016.

- **TODO**
- Zig software foundation
  - only no strings attached donations
- It is often mentioned as a successor to C.

2.2. Toolchain

Installation was as simple as downloading the release tar archive, extracting the toolchain, and symlinking the binary onto a `$PATH`. There is also a community project named `zigup` 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` file. The configuration is written in zig itself and thus avoids a separate language just for the build system. The toolchain itself is also a C/C++ compiler which allows zig code to directly interoperate with existing C/C++ code.

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`.

The zig community also provides a language server named `zls`, which worked out of the box in neovim. There were some issues with type inference, and completion of member functions of generic types. And `zls` reporting no errors when the zig compiler would.

2.3. 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 arbitrary sized integers like `i27`, or `u3`, up to a maximum bit-width of `65535`.

2.4. Arrays and slices

Zig arrays 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 range, returning a slice 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.

```
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 or slices. Most commonly this is used for null-terminated strings:

```
const string: *const [32:0]u8 = "this is a null terminated string";
const fat_pointer_string_slice: []const u8 = string[10..];
const null_terminated_string_slice: [:0]const u8 = string[10..];
const null_terminated_string_ptr: [*:0]const u8 = string[10..];

std.debug.print("size: {}, '{s}'\n", .{@sizeOf([]const u8),
fat_pointer_string_slice});
std.debug.print("size: {}, '{s}'\n", .{@sizeOf([:0]const u8),
null_terminated_string_slice});
std.debug.print("size: {}, '{s}'\n", .{@sizeOf([*:0]const u8),
null_terminated_string_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 also 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 doesn't 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
```

2.5. 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 vari-

able, it's exact type must be specified. When the type of a struct is known, for example when passing it to a function, it's name can be omitted and an anonymous literal can be used:

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

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

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

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

2.6. Control flow primitives

- exhaustive switch statements
  - useful for enums

2.7. Error handling

- errors as values
  - error union explicit or implict
  - try: return on error
  - catch: handle error

In zig there are no exceptions, errors are treated as values. If a function can fail it returns an error union, the error set of that union can either be inferred or explicitly defined.
Similar to rust zig has a try operator that either returns the error from the current function or unwraps the value.

```
// inferred error set
pub fn main() !void {
    const num = try std.fmt.parseInt(u32, "324", 10);
    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;

```

```
    }
}
```

2.8. Defer

There are no destructors in zig, so unlike for example C++ the RAII model can't be used to make objects manage their resources automatically. Instead a common pattern to deal with closable resources is to define an `init` and a `deinit` procedure, which has to be called manually.

When dealing with multiple resources that depend on each other, `deinit` procedures have to be called in reverse initialization order to be properly cleaned up. To make this more ergonomic zig provides `defer` and `errdefer` keywords, which allow deferring cleanup code. `defer` runs code when the value goes out of scope. If multiple defer statements are defined, they are run in reverse declaration order:

```
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();
}
```

`errdefer` runs code only when an error is returned from the scope, this can be useful when dealing with multiple steps that can fail and intermediate resources need to be cleaned up.

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

If the function succeeds the file is returned from it, so it shouldn't be closed. If it fails in a later stage 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 doesn't 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 a `deinit` procedure that can be `defer`red to deallocate the used memory.
In debug mode zig keeps track of allocations and detects memory leaks.

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` or `const` 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 isn't 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.

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

```
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;
    }
    var a = 1;
    var b = 1;
    for (1..n) |_| {
        const c = a + b;
        a = b;
        b = c;
    }
    return b;
}
```

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't 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 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 std library.

2.11. SIMD

In addition to the automatic vectorization of code that the LLVM optimizer does, zig also provides a way to explicitly define vector operations that will compile down to target specific SIMD operations.

```
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 std library of zig is quite minimal. Neither the zig language itself, nor the std library provide a datatype for strings. String literals are represented as byte slices (`[]const u8`), which allows using the whole range of `std.mem.*` functions to operate on them. There is no unicode support. This is done intentionally, and user space libraries can implement their own string datatypes if necessary (TODO: link github issues).

- somewhat immature ecosystem
  - missing regex library
  - async not available in `0.11` self-hosted compiler

3. Development process

Since the scope of the program was predetermined, the main focus was on performance.

The zig std library provides `IterableDir`, an iterator for iterating a directory in a depth first manner, but unfortunately that approach doesn't allow filtering of searched directories. To overcome that limitation I mostly copied the std library function for iterating directories and modified it slightly to allow filtering out hidden directories.

There are some beginnings of regex libraries written in zig, but they are still in their infancy, and aren't feature complete. So I decided on using the rust regex library (`rure`) through it's C API, since it's a standalone project without tethers to a standard library, and reasonably fast.
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 isn't 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 bug (TODO link github issue).

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);
```

3.1. Single threaded optimization

3.1.1. Line by line searching

To keep it simple the first implementation, read the whole file into a single buffer and ran a compiled regex pattern match on every line. This was done using a regex iterator from the `rure` crate.

3.1.2. Whole text searching

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

3.1.3. 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 was done using a fixed size buffer which would only load 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 to calculate line numbers, and the I discovered the `rure` function that searches the text directly I decided to once again search each line individually, instead of the whole text.

3.1.4. Whole text searching with fixed size buffer

After further investigation it turned out that the overhead of searching each line didn't just come from the `rure` iterator, but some special regex patterns introduced large overhead anyway when starting the search. One example was the word character pattern `\w`, which respect possibly multi-byte unicode characters. Since the `rure` library uses a finite automata (state machine), matching multiple word characters results in an exponential explosion of states. Disabling unicode during the regex pattern compilation significantly improved 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 pattern matches that spanned multiple lines. If a match is found that spans multiple lines an additional search is run only on the fist matched line, if this succeeds too only this match is highlighted and printed.

3.2. Parallelization

At this point most easy wins in single threaded optimization were off the table, so the next performance gain would be using multiple threads. The things that take up the largest portion of time are accessing the file system, and searching the text.

Parallelization was implemented using a thread pool of search workers that would receive file paths using a atomically synchronized message queue. The directory walking remained mostly the same apart from searching files adhoc, they were now sent through the message queue.

The output of multiple threads now had to be synchronized so that lines printed 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 synchronization. Once the buffer is full access to stdout is locked for other threads until the file is fully searched, but other workers can still access their thread-local output buffers.

Text searching had been parallelized the search workers were now emptying the message queue too quickly, so walking the file system with multiple threads was up next.
This was heavily influenced by the rust `ignore` by the same author as, and also used in ripgrep. A thread pool of "walkers" is used to search multiple directories simultaneously in a depth first manner to reduce memory consumption. A walker tries to pop of a directory iterator of a shared atomically synchronized stack, by blocking until one is available. Once it receives a directory it iterates through the remaining entries enqueueing any files encountered. If it encounters a subdirectory, the parent directory is pushed back onto the stack and the subdirectory is walked. Once all walkers are waiting for a new directory iterator all directories have been walked completely and the thread pool is stopped.

3.3. 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. `UserArgFlag`s don't require a value and are just boolean toggles. `UserArgValue`s require a value, for

example a number, to be specified after them. `UserArgKind` is a tagged union that either contains 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,
    NoHeading,
    IgnoreCase,
    Debug,
    NoUnicode,
    Help,
};
```

All user args 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.

3.4. 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 fine, bug 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 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) {
```

```
13        Hidden,
14        FollowLinks,
15        Color,
```

4. Conclusion