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

- manual memory management
  - allocators have to be manually passed
  - deallocate by `defering` `deinit` procedures
  - memory leaks are automatically detected in debug mode
- minimal standard library
  - no unicode string support, only functions for operating on slices e.g. std.mem
  - utf-8 string libraries are 3rd party
- exhaustive switch statements
  - useful for enums
- generics are just comptime functions operating on types
- inferred struct literal type `.{}`
- if enum type is known the type can be omitted, `.variant` is sufficient
- errors as values
  - error union explicit or implict
  - try: return on error
  - catch: handle error
- somewhat immature ecosystem
  - missing regex library
  - async not available in `0.11` self-hosted compiler

2.1. Introduction

Zig is a general-purpose compiled systems programming language. Andrew Kelley 2015/2016

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`.

The compiler is a single executable called `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 from zig.

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 called `zls`, which worked out of the box in neovim. There were some issues with type inference, and completion of member functions of generic types.

2.3. Integers

Zig has concrete integer types with a predetermined size regardless of platform. Common ones are:

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

But it also allows defining arbitrary sized integers like `i27`, or `u3`.

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

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

2.4. Arrays and slices

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 includes 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("{d}\n", .{slice.*});
```

The output prints:

```
{ 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});
```

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 condition 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. As a result 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 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. Eco system

Compared to C++, Java, or Rust the std library of zig is quite minimal. Neither the zig language itself, nore the std library provide a datatype for strings. String literals are represented a byte slices (`[]const`

u8), which allows using the whole range of `std.mem.*` functions to operate on them. There is no unicode support. There are some beginnings of people implementing regex libraries, but none that are ready to be used right now. So I decided on using the rust regex library (`rure`) through it's C API, since it's a standalone project and not part of some standard library. 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 });
```

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.

2.6. 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 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,

```

```
10      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) {
      Hidden,
      FollowLinks,
      Color,
```

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.

TODO: rust-regex crate

3.1. Single threaded optimization

3.1.1. Line by line matching

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 matching for performance

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 matching 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 matching for performance 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. TODO: mention ripgrep influence TODO: Walking the file tree uses a shared stack for depth first searching

4. Conclusion