Seminar Programmiersprachen im Verlgeich - Zig

Tobias Schmitz

Contents

Introduction

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

It is often mentioned as a successor to C.

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.

Installation was as simple as downloading the release tar archive, extracting the toolchain, and symlinking the binary onto my `PATH`. 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.

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
```

Language

- build system
  - same language for build-system
  - build.zig is generated
  - good C interoperability
- 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
- slices
  - pointer and length by default: []u8
  - sentinel terminated, for example null terminated: [*0]u8
  - exclusive ranges for slicing: my_slice[0..2]
- somewhat immature ecosystem
  - missing regex library
  - async not available in `0.11` self-hosted compiler
- bug in `0.11` compiler when building in release mode with enum/union tags
- unclear crash messages, even in debug mode
  - just memory addresses
  - this is a bug when linking libc

Development process

Since the scope of the program was pre-determined, 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.

To keep it simple the first implementation, read the whole file into a single buffer.

Single threaded optimization

Line by line matching

Whole text matching for performance

- to avoid allocating iterator for every line
- then avoid using iterator anyway

Line by line matching with fixed size buffer

- no multiline matches
- easier to implement
- lines need to be iterated anyway for line numbers

Whole text matching for performance with fixed size buffer

- the rust regex `find` functions has extremely high startup overhead on unicode word-character patterns
- avoided to some extend by searching the whole text buffer instead of just line by line slices

Parallelization

- Two worker thread pools
  - One for walking the file tree
  - One for searching files
- Walking the filetree uses a shared stack for depth first searching
- A shared queue is used to pass to be searched files
- A shared sink synchronizes writing to stdout

Conclusion