CS 131 Homework 6

Ryan Riahi

March 2021

Abstract

In this report I will evaluate the programming language Zig as a candidate for building a new application for Haversack Inc. This new application is called SecureHEAT and uses inexpensive cameras to monitor the faces of humans, calculate their facial temperatures, and adjust the temperature accordingly. This application can save the company a lot of money, but there are security concerns. Since this application will be put into use in organizations that require high security, we have opted away from C/C++ and thus, I will look into Zig as a potential language to address these concerns and effectively build this application.

1 Introduction

Zig is a low level programming language that first appeared in 2016 and, according to ziglang.org, is a "general-purpose programming language and toolchain for maintaining robust, optimal, and reusable software". It was made to compete with and improve upon C. I will dive into the details of Zig 0.7.1 and evaluate features of the programming language, such as its memory management, type checking, compilation, multi-threaded behavior, and performance. Overall, we desire the software written by Zig to be simple, stripped down, and capable of interfacing with low level hardware.

2 Memory Management

In Zig, programmers must manage their own memory and must handle memory allocation failures. This is similar to C, except that it has a crucial difference: Zig has no default allocator. In C, most memory allocations are made using a call to malloc(); in Zig, however, functions which need to allocate accept an *Allocator parameter that defines the method to allocate memory for them. Thus, programmers can build function specific allocators that tailor their memory allocation specifically to certain use cases. An example of defining an allocator:

```
test "using an allocator" {
    var buffer: [100]u8 = undefined;
    const allocator = &std.heap.
        FixedBufferAllocator.init(&buffer)
            .allocator;
    const result = try concat
            (allocator, "f", "g");
    expect(std.mem.eql(u8, "fg", result));
}
```

In the above example, 100 bytes of stack memory are used to initialize a FixedBufferAllocator, which is then passed to a function.

This is extremely low level and provides for some of the most optimal performance as opposed to other languages. There is no sort of garbage collecting, such as in Java and Python, that would slow the language down, and allocations can be more so optimized than their relative counterparts in C. Zig not having a garbage collector is a huge plus. Garbage collectors are slow and potentially untrustworthy applications. A huge benefit of Zig programmers managing their own memory is that Zig programs are eligible to be used on embedded systems and real time software. Zig programs can be run as freestanding programs with no separate operating system due to its low-level nature, especially with respect to its memory management. A disadvantage, however, is that the programs are potentially less reliable and could be prone to programmer error. Another fault is that creating and defining allocators in Zig is not an easy task and takes away from the programs ease of use and understanding.

3 Types and Type Checking

Zig is a statically typed programming language. This means that Zig performs its type checking at compile time instead of run time. This allows the Zig compiler to detect type errors early and save the program from a potential crash in the future. Without static type checking, even full code coverage tests may be unable to find such type errors. This also allows from the compiler to make more optimizations into the machine code. There is a large benefit of reliability when using statically typed languages. This also provides for a performance advantage over dynamically typed languages. Dynamically typed languages perform the type checks at run time and allow for more flexibility and simplicity in programs. However, this means the interpreter for that language needs to perform type checking at run time which can slow down the program.

One interesting thing about Zig's types is that ordinary pointers cannot be null. Instead, you can use the Optional Type which allows the a value (which is secretly a pointer) to be zero with the compiler checking your work to ensure you do not assign null to something that can't be null. For example, to declare an optional integer:

```
const optional_int: ?i32 = 5678;
```

This is another aspect of Zig that really speaks to its safety and reliability. Lastly, Zig uses both Type Coercion and Explicit Casts for conversions between types, allowing for easy conversions and comparisons of types.

4 Compilation

Zig is a compiled language and this plays an important role in the semantics of the language. Compiled languages offer a big advantage as opposed to interpreted languages, especially in our use case. Compiled languages are generally faster than interpreted ones and offer more error checking before run time. Zig places importance on the concept of whether an expression is known at compile-time. This is done via the comptime keyword which informs the Zig compiler of a compile-time parameter. From the documentation: " A comptime parameter means that: [1] At the callsite, the value must be known at compile-time, or it is a compile error. [2] In the function definition, the value is known at compile-time". This helps to keep the language small, readable, and powerful. Zig also implements generics and compile-time duck typing. For example:

```
  fn max(comptime T: type, a: T, b: T) T {
        return if (a > b) a else b;
}
```

One very cool and interesting aspect of Zig is its capability for inline assembly. For some uses (such as our own), it may be useful to directly control the machine code generated by Zig programs. This can be done via inline assembly which allows the programmer to call a function written entirely in assembly and execute the machine instructions. This has to be tailored to a specific microarchitecture but can prove to be very useful.This is a great way to improve compilation and bypass the OS and run specific machine code into our devices to add security and performance into our SecureHEAT application.

5 Multi-Threaded Behavior

Zig uses asynchronous methods to run its programs concurrently. According to ziglang.org, async functions have "no dependency on a host operating system or even heap-allocated memory. That means async functions are available for the freestanding target". Which is exciting new for building our kind of application. I can imagine our cameras will have to perform a lot of I/O and async functions will be very helpful in terms of performance, and they can be implemented for a freestanding target! Zig implements async functions in a similar way to Python. Quoting from ziglang.org, "The Zig Standard Library implements an event loop that multiplexes async functions onto a thread pool for M:N concurrency". Async functions are fairly straightforward to use and will be easily implementable in our programs.

6 Conclusion

Overall, Zig has shown to be a highly secure, easy to use, flexible, fast, and reliable language and I beleive it to be a great fit for SecureHEAT. The pros heavily out weight the cons, of which there are relatively few. Zig's memory management system allows for secure, flexible, and fast programs to be built. It's types, type checking, and compilation create reliability, safety, and performance. And lastly, it's multi-threaded behavior adds yet another layer of performance and compatibility to our programs. A big con is Zig programs being highly sensitive to human error. Being such a low language, human error can go easily unnoticed and cause lots of problems. Zig's pros are hard to beat however, and I would definitely recommend using this language for our application.

7 References

`https://ziglang.org/`
`https://ziglang.org/documentation/0.7.1/`
`https://web.cs.ucla.edu/classes/winter21/cs131/hw/hw6.html`
`https://ziglang.org/documentation/0.7.1/#Targets`