How I learned to love Zig's diagnostic pattern

August 3, 2024

I have a confession to make. I really like zig. I've been playing with it for a couple-ish years now, and I'm finally very close to publishing at least one project of mine that uses it.

One thing was weird to me at first though — I'd gotten used to error handling by returning union types (like `Result` in Rust). But zig's error unions don't carry payloads. Only an error code without any error context.

So I tried to roll my own error union. Zig has tagged unions (like enums in Rust) after all, how hard could it be?

TL;DR: I now see the wisdom in zig's alternative to error union payloads, the *diagnostic pattern*.

The *diagnostic pattern*

The recommended idiom today, the diagnostic pattern, suggests your error-returning function should take an additional parameter which is a pointer to a struct to fill with the error's context.

Suppose we're writing a JSON parser, we probably want to store the position we were at when we encountered an error. We would handle errors from such a parser like this:

```
const Diagnostic = struct {
    position: usize = 0,
};

test "good json for normal zig errors" {
    var diagnostic: Diagnostic = .{};
    _ = try parseJson(std.testing.allocator, "[1, 2]", &diagnostic);
}

test "bad json for normal zig errors" {
    var diagnostic: Diagnostic = .{};
    const result = parseJson(std.testing.allocator, "[1, 2, invalid]", &diagnostic);
    // we must check there was an error before we can use the diagnostic
    try std.testing.expectEqual(result, error.InvalidToken);
    try std.testing.expectEqual(diagnostic.position, 7);
}
```

Some people don't like it because the disconnectedness of the parameter to the return value makes it easy to forget to use the pattern at all, or just forget to populate the diagnostics. The zig standard library has historically forgotten to return error context, potentially because of

this (if zig is old/stable enough to have a "history").

So, can we get error context without adding a parameter? Do we want to? We could "just" use zig's language support for tagged unions to make our own union with one (or more) error states, and return that.

But then you miss out on using zig's builtin `try`, `catch`, and `errdefer` to handle errors. In languages with more hidden control flow, for example Rust, you can extend the `?` operator to do ergonomic error handling with user-defined types.

In practice I think lacking `try`/`catch` turns out to be liveable, but, more on `errdefer` later.

Here we go:

```
pub fn Result(comptime R: type, comptime E: type) type {
    return union(enum) {
        ok: R,
        err: E,
    };
}

fn parseJsonResult(alloc: std.mem.Allocator, json_src: []const u8) Result(JsonValue, Diagnostic) {
  //...
}

test "good json for our result type" {
    const result = parseJsonResult(std.testing.allocator, "[1, 2]");
    try std.testing.expect(result == .ok);
}

test "bad json for custom result type" {
    const result = parseJsonResult(std.testing.allocator, "[invalid]");
    try std.testing.expect(result == .err);
    try std.testing.expectEqual(result.err.position, 1);
}
```

So, idiomatic `try` and `catch` become more cumbersome.

```
fn useParseJson(alloc: std.mem.Allocator, diagnostic: ?*Diagnostic) void {
  const json = try parseJson(alloc, "[]", diagnostic);
}

fn useParseJsonResult(alloc: std.mem.Allocator) Result(void, Diagnostic) {
  // you can also use a switch for no extra variable but imo its ugly
  const result = parseJsonResult(alloc, "invalid_json");
  const json = if (result == .ok) result.ok else return result;
  return Result(void, Diagnostic){.ok={}};
}
```

`errdefer`

`errdefer` is worse, but might get better in the future.

Remember that `errdefer` is important in cases such as when you allocated something to return to the caller to own, but now they receive only the error and you need to destroy your partial successful result to not leak resources.

```
fn errdeferParseJson(alloc: std.mem.Allocator, diagnostic: ?*Diagnostic) std.ArrayList([]const u8) {
  const result = std.ArrayList([]const u8).init(alloc);
  errdefer result.deinit();
  const json = try parseJson(alloc, "[]", diagnostic);
  // ...use json to build the result somehow
  return result;
}

fn errdeferParseJsonResult(alloc: std.mem.Allocator) Result(std.ArrayList([]const u8), Diagnostic) {
  var ok = std.ArrayList([]const u8).init(alloc);
  var result = Result(std.ArrayList([]const u8), Diagnostic) = .{.err = .{}};
  defer if (result == .err) ok.deinit();

  const json = switch (parseJsonResult(alloc, "[]")) {
    .ok => |o| o,
    .err => |e| {
      result = .{.err = e },
      return result;
    },
  };

  result = Result(std.ArrayList([]const u8), Diagnostic) = .{.ok = ok};
  return result;
}
```

Everywhere that you return an error, you must manually first set the `result` variable and then return it, to be sure that our `errdefer` replacement, (`defer if (result == .err) ...`) takes effect.

That's risky. As the code base changes, someone could totally forget that unenforced contract and return an error directly, leaving `Result` in an accidentally valid state and causing a leak.

Also that's a lot of boilerplate. Maybe the diagnostic pattern isn't so bad?

The future for `errdefer`'ing hand-rolled error unions

In the future, we may get a language builtin to access the return value directly. This may have the side-effect of allowing us to access our returned value during a `defer` statement, eliminating the need to manually make sure we're always returning the correct variable.

We would end up with the much nicer:

```
fn errdeferFutureParseJsonResult(alloc: std.mem.Allocator) Result(std.ArrayList([]const u8), Diagnostic) {
  var result = std.ArrayList([]const u8).init(alloc);
  defer if (@return() == .err) result.deinit();

  const json = switch (parseJsonResult(alloc, "[]")) {
    .ok => |o| o,
    .err => |e| return .{.err = e },
  };

  return Result(std.ArrayList([]const u8), Diagnostic){.ok = result};
}
```

I would say that's pretty reasonable all of a sudden.

Which is better?

Well, now that we've thoroughly compared the code for both options, which wins?

It looks to me like the diagnostic pattern is surprisingly simple, low on boilerplate, and integrates well with the language today. The main advantage of the `Result` type is it always requires full initialization in the error case which makes it less prone to forgetting to set the error context.

Another advantage of the diagnostic pattern that is not immediately apparent is that it fits well with exported functions. `extern` unions can't be automatically tagged so returning them in a C API takes some effort. But just slap an `extern` on the Diagnostic struct and you can pretty straight-forwardly export any functions we wrote already using the diagnostic pattern.

I felt this recently when I moved a project I'm working on from my hand-rolled results to the diagnostic pattern, and the exported C API got much simpler. Especially with zig and its strong C interop, I find myself exporting a crafted C API often so I find this useful.

The result pattern could fit well if the language underwent some changes but I don't see it as worth the complexity after trying both.

I am much happier with the diagnostic pattern now.