Structured diagnostics for the OCaml compiler

Compiler error messages are one of the basic block of a language developer experience. However, many other tools –from language servers to build systems– participate to this experience. Thus, compiler diagnostics ought to be understandable not only by humans but also tools. How do we add machine-readable diagnostics to the OCaml compiler?

1 Motivation

In strongly typed languages, like OCaml, error messages and other compiler diagnostics are a core part of the language developer experience. However, this core ought to be weaved into a well-integrated experience between the language compiler, the build system, and their editor of choice.

Moreover integrating plain text compiler diagnostics in a language server can only go so far. Consider for instance a moderately complex OCaml error message. First, we may start with quoting the source code

```
8 |     module U = F(struct type x end)(B)
```

Then we have a high-level description of the type error

```
Error: This application of the functor F is ill-typed.
       These arguments: $S1 P.B do not match these parameters: functor (X : x) (B : b/2) -> ...
```

which, in this case, is followed by a argument-by-argument analysis of the ill-typed functor application

```
1. Module $S1 matches the expected module type x
2. Modules do not match: P.B : b is not included in b/2
```

This is a lot of structure that a graphical editor could exploit, if it could discover this structure without haphazardly parsing the compiler output.

2 Structured logging as incremental algebraic data type construction

To better cooperate with other tools, the compiler itself should provide machine-readable structured diagnostics. This idea has been adopted([1],[2],[5],[4]) or is under consideration ([3],[7],[6]) in many compilers.

However, is the file format that important? What really matters is that the data is well-structured, in other words well-typed. And with GADTs, we can create an embedded versioned algebraic type system for diagnostics:

```
type 'a typ =
  | Int: int typ | Doc: doc typ | Option: 'a typ -> 'a option typ
  | Sum: 'a def -> 'a sum typ | Record: 'id def -> 'id record typ | ...
```

From that description, we will show how to build various printers for diagnostics, enforce a versioning policy, provide logging functions that can replace transparently Format printers and more.

3 A document type for structured text

Looking more closely to this fragment of error message,

> 2. Modules do not match: P.B : b is not included in b/2

it has been formatted by the compiler taking in account that the terminal width is 80 and it supports ANSI escape. But when a tool reprints this error message, we are probably not printing to the same terminal.

Thus, the compiler should let formatting decisions to downstream tools. But how do we update the compiler text format in a lightweight way? We shall present how the OCaml compiler has done so by using an alternative Format implementation [1].

4 Covariance for maintainability

Another requirement for structured compiler output is the ability to guarantee a sufficient level of backward compatibility to other developer tools.

We shall show how by weaving a notion of sequence of versions through the creations and deletions of both record fields and variant constructors, we can ensure that minor updates are covariant: a minor update may only create record fields or delete variant constructors. The reverse breaking changes can then be relegated to major updates.

Similarly, with our versioned eDSL for compiler diagnostics, we can check at runtime that a diagnostic is well-typed for a given version.

5 Conclusion

Using GADTs as a eDSL for a subset of versioned algebraic data types, we can implement structured logging as an incremental construction of a compiler diagnostics seen as an algebraic data type. In turn, this architecture frees us from choosing a file format for compiler diagnostics, while providing a well-defined type for compiler diagnostics and enforcing a given versioning policy.

References

[1] The Rust Core Team. *Announcing Rust 1.12*. 2016. URL: https://blog.rust-lang.org/2016/09/29/Rust-1.12.html.

[2] Malcolm David. *Usability improvements in GCC 9*. 2019. URL: https://developers.redhat.com/blog/2019/03/08/usability-improvements-in-gcc-9.

[1] https://github.com/ocaml/ocaml/13169

[3] Alfredo Di Napoli. *The new GHC diagnostic infrastructure*. 2021. URL: https://well-typed.com/blog/2021/08/the-new-ghc-diagnostic-infrastructure.

[4] Christopher Di Bella. *WIP [clang] adds capabilities for SARIF to be written to file*. 2023. URL: https://reviews.llvm.org/D145284.

[5] Brand Sy. *Structured Diagnostics in the New Problem Details Window*. 2023. URL: https://devblogs.microsoft.com/cppblog/structured-diagnostics-in-the-new-problem-details-window.

[6] Clements Austin. *cmd/go: go build -json*. 2024. URL: https://github.com/golang/go/issues/62067.

[7] Eisenberg Richard. *-ddump-json is underspecified*. 2024. URL: https://gitlab.haskell.org/ghc/ghc/-/issues/19278.