GSoC proposal: A language server for Dhall

Student: Frederik Ramcke, Mentor: Gabriel Gonzalez

Abstract

The aim of the proposed project is to improve support for the Dhall language in mainstream editors via the *Language Server Protocol* standard.

1 Introduction

Dhall is a programmable configuration language that first originated as a Haskell package. While it is already being quickly picked up both by industrial and hobby users, we hope to make it even more approachable by improving its support in mainstream editors.

The Dhall configuration language Dhall is a programmable *configuration language* that brings principles from the world of *strongly typed functional programming* into the less academic world of configuration files. More specifically, Dhall is a polymorphic lambda calculus *without general recursion*—all Dhall programs are guaranteed to terminate! In a sense this makes Dhall a very restricted language, that gives strong guarantees at the expense of expressiveness; this trade-off turns out to be extremely desirable in a configuration language.

As a configuration language Dhall's design includes a few domain specific features. As an example, while Dhall has no notion of module, Dhall supports referencing (i.e. importing) arbitrary expressions by path or URL. Dhall also allows us to annotate such import statements with the expected hash of the imported expression, thus "freezing" the import.

Finally, Dhall's lack of general recursion together with its formal semantics means that Dhall code satisfies a property called *strong normalization.* In practice this means that any Dhall program/expression is guaranteed to terminate, and moreover that any expression has a unique *normal form*; for example any expression of type Natural is guaranteed to evaluate to a natural number (like 42 or 1337). Note that this holds for expressions of *any* type, for example the normal form of the function $\lambda(b : \text{Bool}) \to b \,\&\&\, b$ is $\lambda(b : \text{Bool}) \to b$.

IDE support Editor integration, from basic syntax highlighting to more advanced features like "jump to definition", makes the life of software developers easier. In the case of Dhall, the hope is to make the language even more attractive to mainstream users by providing a polished development experience.

The design of the Dhall language further allows us to offer some very unique features, like for example reducing any expression to a canonical *normal form* free of indirection and obfuscation (which is useful when auditing code from untrusted sources), or inlining imports.

The project This proposal aims towards developing language integration for the Dhall language into most of the mainstream editors (e.g. Atom, VSCode, Emacs etc.) by leveraging the *Language Server Protocol* (LSP) standard. The LSP defines a unifying framework for interacting with editors, allowing us to reuse the same *language backend* for different IDEs.

Thanks to PanAeon, another contributor, a basic prototype implementation of a Dhall language server and accompanying VSCode client already exists; the proposed project will build on this existing codebase. The deliverables are therefore going to be new releases of the language server and matching VSCode client.

2 Benefit to the community

The most immediate benefit this project will provide is an improvement to the infrastructure surrounding the Dhall language, therefore improving the coding experience for its users.

Additionally, a fully featured Dhall language server will serve as a useful example of how to improve editor support for other languages. This is particularly relevant to the Haskell community—after all Haskell is the language of choice for parser and compiler writers.

3 About me

I am Frederik, a master's student of computer science at the *Chalmers University of Technology* in Göteborg, Sweden. My academic interests lie in type theory and logic, and in my free time I try to climb (as in rock climbing) as much as possible.

As part of my thesis project I am mainly working with Agda, a dependently typed language for programming and proofs that shares some similarity with Haskell. For personal projects I prefer working with Haskell (who doesn't), but I also have experience with C/C++, Java, and similar imperative languages.

My academic background being type theory means that I am quite comfortable with the theoretical background of the Dhall language; parsers and type checkers are also not unfamiliar to me. My experience with writing "user-facing" code is however limited; in this respect I am particularly looking forward to learning a lot through the project!

4 Project Scope

The aim of this project is to deliver a polished product, that is, any of the contributed features should be immediately useable and useful to the end user. To that end I will:

Focus on a single editor I will focus on VSCode as the target editor for the scope of this project. Though the language server protocol promises to unify editor integration, in practice we still need to supply a "client" plugin for each supported editor. Here we choose "depth over breadth" and leave further editor support as future work.

Leverage the Language server protocol I will try to stay within the core features of the LSP standard that are supported without having to provide additional client-side code. This way I avoid having to deal with Javascript more than absolutely necessary, and I also minimise the amount of bespoke code that will have to be reimplemented for every additional editor down the line.

Start from existing prototype PanAeon (GitHub user name) already set up the infrastructure for a basic language server and client (the language server currently supports error highlighting and formatting). By building on existing infrastructure I will hopefully be able to be ramp up my productivity very quickly.

4.1 Detailed Features

The following is a detailed list of features that I intend to implement. They are ordered in increasing complexity, corresponding to the order in which I will implement them.

Linting The existing prototype already allows the user to **format** Dhall code. I will polish this feature by

- allowing the user to **lint** the code as well
- providing feedback to the user, confirming that linting happened or reporting linting errors
- adding an option to format/lint Dhall files on saving them

This feature is simple to implement since it just makes existing functionality of the Dhall package available to the user (the Dhall Haskell package exports an API both for linting and formatting).

Type Errors This is the second existing feature of the prototype. What needs to be done:

- making error ranges more precise (they currently include trailing whitespace)
- add an option to provide Dhall's detailed error messages

Type on hover On hovering over an identifier present its type. On the backend side this will involve having to work with the Dhall AST and type checker. On the client side this should be immediately supported by the "Hover request" functionality of the LSP.

This is the first novel feature that I will contribute to the language server. In my research leading up to this proposal I considered this feature to guide me in understanding the Dhall codebase, and I found that the implementation will boil down to a relatively simple AST traversal.

Annotate let bindings Add a right-click command to annotate a "let" statement with its type, i.e. transforming $\textbf{let}\, x = t$ into $\textbf{let}\, x : A = t$ by inferring the type of the body t.

This will involve inserting type information into the AST and rendering it back to the user; by always running the code formatter after AST changes we can avoid some complexity here. On the other hand, I will have to add code to the client to add the right-click functionality.

Imports Add right-click commands to

- open an imported file
- "freeze" an import statement, i.e. to annotate it with the cryptographic hash of its content
- inline an import statement, i.e. replacing an external reference with its contents

This will build on the work done implementing the previous feature. Note that the Haskell implementation of the Dhall language already exposes the freezing and inlining of imports; what remains is to expose these features through the LSP.

Normalisation Allow the user to select an expression and

- α-normalise the expression, replacing variable names with their DeBruijn indices
- β-normalise the expression, i.e. resolving function applications etc.

The main difficulty will lie in implementing the selection of expressions on the client side (this is not immediately supported by the LSP) and mapping these to subtrees of the AST.

5 Proposed Timeline

Before 27 May (*Community Bonding period)* I will continue to familiarise myself with the following:

- the specification of the Language Server Protocol
- the existing client-side code (vscode-dhall-lsp)
- the haskell-lsp project, which supplies the infrastructure for our plugin for interacting via the LSP
- the existing language server prototype (dhall-lsp-server)

I will also announce my work on the project at discourse.dhall-lang.org; this will be my primary way of interacting with the community (outside of Git commits and pull requests). Later, I will publish reports on each finished feature on the forum and use that to gather feedback.

27 May–17 June *(Warm up)* Implement the proposed changes to the existing functionality of the prototype (polishing linting and displaying of error messages). Write/gather documentation on how to install the language server and how to use the existing features. The main purpose of this is for me to get comfortable with the the codebase.

17 June–1 July (First milestone) Implement the "type on hover" functionality. This is the first novel contribution.

1 July–15 July (Second milestone) Implement the "annotate lets" feature. This is the first feature that will need nontrivial changes to the VSCode client.

15 July–29 July (Third milestone) Implement freezing, inlining, and following imports. Builds on the "annotate lets" feature.

29 July–12 August (Fourth milestone) Implement the α/β normalisation feature.

12 August–26 August (Final Stretch) I leave a buffer of about two weeks to absorb any unforeseen delays. If time remains I will use this time to review the documentation I produced in the previous months and improve it further.