ChatLSP: Making Language Servers Do More for Statically Contextualizing Large Language Models

June Hyung Kim
University of Michigan
Ann Arbor, USA

Abstract

Developer tooling is crucial for developer productivity. Language Server Protocol, in particular, allow language features to be communicated to editors without having to rely on hacky editor magic. However, current implementations of the protocol do not adequately support the meteoric rise in demand for code completion via LLMs. LLMs are prone to hallucination, and existing methods such as exhaustive retrieval and RAG are limited by token size and inconsistency. Approaching this with a language semantics-based retrieval can have benefits over unstructured approaches, but LSP currently do not expose an interface for static retrieval. We propose a conservative addendum to the Language Server Protocol, ChatLSP, to address this issue.

ACM Reference Format:
June Hyung Kim. 2024. ChatLSP: Making Language Servers Do More for Statically Contextualizing Large Language Models. In . ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction and Background

One of the biggest advancements in the tooling ecosystem is the Language Server Protocol [8], which provides a unified interface for linking language and editor features. Recently, we've seen an explosion at the juncture of editor tooling and AI, with GitHub Copilot [3] leading the charge, promising to help developers. However, these AI tools are also prone to hallucination, suggesting statically incorrect code or statically correct code that exhibits runtime errors. Ameliorating these negative effects can be done by feeding the LLM more information about the codebase. Because costs increase with token count [2, 6], methods such as RAG [4] are commonly used. This is not ideal, though, as it only grabs seemingly relevant tokens based on imperfect heuristics. Treating code as more than mere tokens by using static retrieval cleanly links the completion site to type and value definitions elsewhere in the codebase [1].

We define static retrieval as the action of retrieving expected types, relevant types, and relevant headers.

```
type Amount = number;
type Currency = Dollar | Euro;
type Transaction = [Amount, Currency];
const createAmount = (n: number): Amount => {
    return Math.round(n * 100) / 100;
}
const withdraw = (n: number): Transaction => ??
```

Figure 1. Example TypeScript Program

Fig. 1 shows an example TypeScript Program for demonstration purposes. The *??* represents a typed hole, an incomplete portion of the code. Similar concepts exist in languages such as Agda [9] or Haskell [10].

An expected type is the type of the hole. In Fig. 1, the expected type is *type withdraw = (n: number) => Transaction*. Relevant types is the list of types that may be related in any way to the expected type. The expected type can comprise components that refer to other types, so we retrieve those references. In Fig. 1, the relevant types are the following:

- *type withdraw = (n: number) => Transaction*
- *type Transaction = [Amount, Currency]*
- *type Amount = number*
- *type Currency = Dollar | Euro*
- Types that *Dollar* and *Euro* refer to.

Relevant headers is the list of functions that we can call to return the expected type, or parts of it. In Fig. 1, the only relevant header is *createAmount*.

Determining what headers are relevant can be done through two steps: generating a list of target types, and finding headers whose types are consistent with any of the target types. Criteria for target types are as follows:

- The type of the hole itself.
- If the hole is an arrow type, the return type.
- If the hole is a product type, the component types.
- Compound type definitions (cases where the return type is a product or a product has an arrow type).

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

There are many ways to determine type consistency, as this depends on the language implementation. One example would be to use a checker function *const checker = (h: HTyp): TTyp => return h*, and check that substituting *HTyp* for the header's type and *TTyp* for the target's type does not return a static error when compiled.

2 Limitations of Modern Developer Tooling

Current implementations of developer tooling such as LSPs and static analysis tools do not operate at a granular type-based level capable of the aforementioned static retrieval.

2.1 Limitations of the Language Server Protocol

Language servers are capable of retrieving static information from a codebase. Particularly, modern language servers often have the ability to retrieve the expected type of an expression, the relevant types, and the relevant function headers. However, the current Language Server Protocol [7] does not provide an interface to capitalize on this. Because of this, static retrieval must be done by hacking together different LSP methods in a recursive algorithm.

- Get the current cursor location.
- Get the enclosing declaration.
- Get the type of the expression inside.
- Step into the expression's component types, and recurse. For example, in Fig. 1, if the expression at hand is *Currency*, recurse on *Dollar* and *Euro*.

Clients can request one specific method, but cannot request for a series of methods in a programmatic manner. Such action would require a custom method that the client can call once, and the language server to execute the tasks. Our research showed that type and header extraction is not implemented yet. Given the importance of extracting static information and the capabilities of language servers, we believe that this is a missed opportunity.

2.2 Limitations of Other Static Analysis Tools

Another popular static analysis tool is CodeQL [5] by GitHub. CodeQL first generates a database over a repository, then gives the developers the ability to write queries on specific elements, AST nodes, and more. In principle this is the ideal way to extract information. Code, by definition, is more than just text and cursor positions - it is semantic data, and querying for specific data inherently makes sense.

CodeQL, however, is not without its flaws. The database is created over a snapshot of the codebase, so it does not incrementally expand with the addition of new code - it must be re-created every time a new change occurs. Furthermore, the overhead is non-negligible - creating a database takes a considerable amount of time, and developers must also write separate queries in CodeQL's own domain-specific language QL. We believe that developers should be able to do relatively simple things such as static retrieval without writing complicated queries.

3 Extending the Protocol

An ideal system should have the the following qualities:

- Extract static information, particularly the expected type, relevant types, and relevant headers.
- Fast, actions should be completed in milliseconds.
- Easy to use, and developers who use an editor should not have to learn thrid-party tools to reap the benefits of static retrieval.
- React to future code changes.

Given these limitations, many third-party static analysis tools are out of question. Language servers are still incredibly attractive options - they benefit from being extendable and capable of accomplishing tasks in milliseconds. Therefore, instead of reinventing the wheel, we decide to propose a conservative addendum to the Language Server Protocol. This comprises five new methods.

- *textDocument/expectedType*
- *textDocument/relevantTypes*
- *textDocument/relevantHeaders*
- *textDocument/llmCompletion*
- *textDocument/llmCompletionResolveErrors*

textDocument/expectedType returns the expected type of an incomplete expression.

textDocument/relevantTypes returns a list of strings, each representing a type definition that may be relevant at a given cursor location. The returned list contains components that can be used to build up the expected type.

textDocument/relevantHeaders returns a list of strings, each representing a function header that we can call to return the expected type, or part of it, at a given cursor location.

textDocument/llmCompletion works similarly to the previously existing completion method, but uses LLM to complete the code, akin to GitHub copilot. The result is immediately visible in the editor, and users will be able to tab through different completions.

textDocument/llmCompletionResolveErrors will attempt to resolve incorrect completions by using error rounds, iteratively fixing the generated code by replying with static errors and prompting for a corrected version.

4 Future Work

We are in the process of creating a Visual Studio Code extension that uses this extended language server. We hope to test this with developers and conduct a study on developer productivity versus current implementations of LSP and Copilot.

Acknowledgments

To Dr. Cyrus Omar and Andrew Blinn for the late nights.

References

[1] Andrew Blinn, Xiang Li, June Hyung Kim, and Cyrus Omar. 2024. Statically Contextualizing Large Language Models with Typed Holes. (2024).

[2] Yiran Ding, Li Lyna Zhang, Chengruidong Zhang, Yuanyuan Xu, Ning Shang, Jiahang Xu, Fan Yang, and Mao Yang. 2024. LongRoPE: Extending LLM Context Window Beyond 2 Million Tokens. arXiv:2402.13753 [cs.CL]

[3] Nat Friedman. 2021. Introducing GitHub Copilot: your AI pair programmer. https://github.blog/2021-06-29-introducing-github-copilot-ai-pair-programmer/

[4] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Qianyu Guo, Meng Wang, and Haofen Wang. 2023. Retrieval-Augmented Generation for Large Language Models: A Survey. *CoRR* abs/2312.10997 (2023). https://doi.org/10.48550/ARXIV.2312.10997 arXiv:2312.10997

[5] GitHub. [n. d.]. codeql. https://codeql.github.com

[6] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. 2022. Training Compute-Optimal Large Language Models. arXiv:2203.15556 [cs.CL]

[7] Microsoft. [n. d.]. Language Server Protocol Specification - 3.17. https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/

[8] Microsoft. [n. d.]. Official Page for Language Server Protocol. https://microsoft.github.io/language-server-protocol/

[9] The Agda Team. [n. d.]. Lexical Structures. https://agda.readthedocs.io/en/v2.6.0.1/language/lexical-structure.html

[10] The GHC Team. [n. d.]. 7.14. Typed Holes, The Glorious Glasgow Haskell Compilation System User's Guide, Version 7.10.2.20151030. https://downloads.haskell.org/~ghc/7.10.3-rc1/users_guide/typed-holes.html