Introduction

Integrated Development Environments (IDEs) and other more lightweight code editors are by far the most used tool of software developers. Yet, improvements of language intelligence, i.e. code completion, debugging as well as static code analysis and enrichment, have traditionally been subject to both the language and the editor used. Language support is thereby brought to IDEs by the means of platform dependent extensions that require repeated efforts for each platform and hence varied a lot in performance, feature-richness and availability. Recent years have seen different works [refs?] towards editor independent code intelligence implementations and unified language independent protocols one of which being put forward by Microsoft - the Language Server Protocol [ref] which is discussed in greater detail in sec. **??**. These approaches reduced the effort of implementing language intelligence from $\mathcal{O}(E \times L)$ to $\mathcal{O}(1 \times L)$ where E stands for the number of *editors* and L for *languages*. As a side effect this also allows for developers to stay in their preferred developing environment instead of needing to resort to e.g. Vim or Emacs emulation or loosing access to other plugins.

Being independent of the editors, the choice of language to implement language servers in lies with the developer. In effect, it is possible for language developers to integrate essential parts of the existing language implementation for a language server. By now the LSP has become the most popular choice for cross-platform language tooling with implementations [langservers and microsoft] for all major and many smaller languages.

Speaking of smaller languages is significant, as both research communities and industry continuously develop and experiment with new languages for which tooling is unsurprisingly scarce. Additionally, previous research [ref], that shows the importance of language tools for the selection of a language, highlights the importance of tooling for new languages to be adopted by a wider community. While previously implementing language tools that integrate with the developer's environment was practically unfeasible for small projects due to the incompatibility between different extension systems, leveraging the LSP reduces the amount of work required considerably.

Problem Definition

Yet, while many of the implementations are freely available as Open Source Software [ref?], the methodology behind these servers is often poorly documented, especially for smaller languages. There are some experience reports [ref: merlin, and others] and a detailed video series on the Rust Analyzer[ref or footnote], project, but implementations remain very opinionated and poorly guided through. The result is that new implementations keep repeating to develop existing solutions.

Moreover, most projects do not formally evaluate the Language Server on even

basic requirements. Naïvely, that is, the server should be *performant* enough not to slow down the developer, it should offer *useful* information and capabilities and of course be *correct* as well as *complete*.

To guide future implementations of language servers for primarily small scale languages the research presented in this thesis aims to answer the following research questions at the example of the Nickel Project[1]:

RQ.1 How to develop a language server for a new language with the abovementioned requirements in mind?

RQ.2 How can we assess the implementation both quantitatively based on performance measures and qualitatively based on user satisfaction?

RQ.3 Do the methods used to answer RQ.1 meet the expected requirements under the assessment developed in RQ.2?

Goals

The goal of this research is to describe a reusable approach for representing programs that can be used to query data to answer requests on the Language Server Protocol efficiently. The research is conducted on an implementation of the open source language Nickel[ˆhttps://nickel-lang.org] which provides the *Diagnostics*, *Jump to ** and *Hover* features as well as limited *Auto-Completion* and *Symbol resolution*. Although implemented for and with integration of the Nickel runtime, the goal is to keep the internal format largely language independent. Similarly, the Rust based implementation should be described abstractly enough to be implemented in other languages. To support the chosen approach, a user study will show whether the implementation is able to meet the expectations of its users and maintain its performance in real-world scenarios.

Non-Goals

The reference solution portrayed in this work is specific for the Nickel language. Greatest care is given to present the concepts as generically and transferable as possible. However, it is not a goal to explicitly cover a problem space larger than the Nickel language, which is a pure functional language based on lambda calculus featuring recursive record types and optional typing.

Research Methodology

What are the scientific methods

Structure of the thesis

[1] https://nickel-lang.org

Background

This thesis illustrates an approach of implementing a language server for the Nickel language which communicates with its clients, i.e. editors, over the open Language Server Protocol (in the following abbreviated as *LSP*). The current chapter provides the backgrounds on

This chapter aims to provide an understanding of the underlying technologies and use-cases of this project.

Language Server Protocol

Rationale

Commands and Notifications

File Notification

Diagnostics

Hover

Completion

Go-To-*

Symbols

Infrastructure as Code

Software Networks

Data oriented languages

Nickel

Gradual typing

Row types

Contracts

Related Work

Discussion

Future Work