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[^https://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 background on the technological details of the project. As the work presented aims to be transferable to other languages using the same methods, this chapter will provide the means to distinguish the nickel specific implementation details.

The primary technology built upon in this thesis is the language server protocol. The first part of this chapter introduces the LSP, its rationale and improvements over classical approaches, technical capabilities and protocol details. The second part is dedicated to Nickel, elaborating on the context and use-cases of the language followed by an inspection of the technical features of Nickel.

Language Server Protocol

Language servers are today's standard of integrating support for programming languages into code editors. Initially developed by Microsoft for the use with their polyglot editor Visual Studio Code[2] before being released to the public in 2016 by Microsoft, RedHat and Codeenvy, the LSP decouples language analysis and provision of IDE-like features from the editor. Developed under open source license on GitHub[3], the protocol allows developers of editors and languages to work independently on the support for new languages. If supported by both server and client, the LSP now supports more than 24 language features[4] including code completion, code navigation facilities, contextual information such as types or documentation, formatting, and more

Motivation

Since its release, the LSP has grown to be supported by a multitude of languages and editors[@langservers [@lsp-website]], solving a long-standing problem with traditional IDEs.

Before the inception of language servers, it was the editors' individual responsibility to implement specialized features for any language of interest. Under the constraint of limited resources, editors had to position themselves on a spectrum between specializing on integrated support for a certain subset of languages and being generic over the language providing only limited support. As the former approach offers a greater business value, especially for proprietary products most professional IDEs gravitate towards excellent (and exclusive) support for single major languages, i.e. XCode and Visual Studio for the native languages for

---

[2]https://code.visualstudio.com/

[3]https://github.com/microsoft/language-server-protocol/

[4]https://microsoft.github.io/language-server-protocol/specifications/specification-current/

Apple and Microsoft Products respectively as well as JetBrains' IntelliJ platform and RedHat's Eclipse. Problematically, this results in less choice for developers and possible lock-in into products subjectively less favored but unique in their features for a certain language. The latter approach was taken by most text editors which in turn offered only limited support for any language.

Popularity statistics[5] shows that except Vim and Sublime Text, both exceptional general text editors, the top 10 most popular IDEs were indeed specialized products. The fact that some IDEs are offering support for more languages through (third-party) extensions, due to the missing standards and incompatible implementing languages/APIs, does not suffice to solve the initial problem that developing any sort of language support requires redundant resources.

This is especially difficult for emerging languages, with possibly limited development resources to be put towards the development of language tooling. Consequently, community efforts of languages any size vary in scope, feature completeness and availability.

The Language Server Protocol aims to solve this issue by specifying a JSON-RPC[^Remote Procedure Call] API that editors (clients) can use to communicate with language servers. Language servers are programs that implement a set of IDE features for one language and exposing access to these features through the LSP, allowing to focus development resources to a single project that is above all unrelated to editor-native APIs for analytics processing code representation and GUI integration. Consequently, now only a single implementation of a language server is required, instead of one for each editor and editor maintainers can concentrate on offering the best possible LSP client support to their product independent of the language.

JSON-RPC

JSON-RPC (v2) [@json-rpc] is a JSON based lightweight transport independent remote procedure call protocol used by the LSP to communicate between a language server and a client.

The protocol specifies the general format of messages exchanges as well as different kinds of messages. The following snippet lst. 1 shows the schema for request messages.

The main distinction in JSON-RPC are *Requests* and *Notifications*. Messages with an `id` field present are considered *requests*. Servers have to respond to requests with a message referencing the same `id` as well as a result, i.e. data or error. If the client does not require a response, it can omit the `id` field sending a *notification*, which servers cannot respond to, with the effect that clients cannot know the effect nor the reception of the message.

Responses as shown in lst. 2, have to be sent by servers answering to any

---

[5]https://web.archive.org/web/20160625140610/https://pypl.github.io/IDE.html

Listing 1 JSON-RPC Request

```
// Requests
{
  "jsonrpc": "2.0"
, "method": String
, "params": List | Object
, "id": Number | String | Null
}
```

request. Any result or error of an operation is explicitly encoded in the response. Errors are represented as objects specifying the error kind using an error `code` and providing a human-readable descriptive `message` as well as optionally any procedure defined `data`.

Listing 2 JSON-RPC Response and Error

```
// Responses
{
  "jsonrpc": "2.0"
  "result": any
  "error": Error
, "id": Number | String | Null
}
```

Clients can choose to batch requests and send a list of request or notification objects. The server should respond with a list of results matching each request, yet is free to process requests concurrently.

JSON-RPC only specifies a message protocol, hence the transport method can be freely chosen by the application.

Commands and Notifications

The LSP build on top of the JSON-RPC protocol described in the previous subsection.

File Notification

Diagnostics

Hover

Completion

Go-To-*

Symbols

code lenses

Shortcomings

Configuration programming languages

Nickel [@nickel], the language targeted by the language server detailed in this thesis, defines itself as "configuration language" used to automize the generation of static configuration files.

Static configuration languages such as XML[@xml], JSON[@json], or YAML[@yaml] are language specifications defining how to textually represent structural data used to configure parameters of a program[6]. Applications of configuration languages are ubiquitous especially in the vicinity of software development. While XML and JSON are often used by package managers [@npm, [@maven], [@composer]], YAML is a popular choice for complex configurations such as CI/CD pipelines [@travis, [@ghaction], [@gitlab-runner]] or machine configurations in software defined networks such as Kubernetes and docker compose.

Such static formats are used due to some significant advantages compared to other formats. Most strikingly, the textual representation allows inspection of a configuration without the need of a separate tool but a text editor and be version controlled using VCS software like Git. For software configuration this is well understood as being preferable over databases or other binary formats. Linux service configurations (files in `/etc`) and MacOS `*.plist` files which can be serialized as XML or a JSON-like format, especially exemplify that claim.

Yet, despite these formats being simple to parse and widely supported [@json], their static nature rules out any dynamic content such as generated fields, functions and the possibility to factorize and reuse. Moreover, content validation has to be developed separately, which led to the design of complementary schema specification languages like json-schema [@json-schema] or XSD [@xsd].

These qualities require an evaluated language. In fact, some applications make heavy use of config files written in the native programming language which gives them access to language features and existing analysis tools. Examples include JavaScript frameworks such as webpack [@webpack] or Vue [@vue] and python package management using `setuptools`[@setuptools].

Despite this, not all languages serve as a configuration language, e.g. compiled languages and some domains require language agnostic formats. For particularly complex products, both language independence and advanced features are desirable. Alternatively to generating configurations using high level languages, this

---

[6]some of the named languages may have been designed as a data interchange format which is absolutely compatible with also acting as a configuration language

demand is addressed by more domain specific languages. Dhall [@dhall], Cue [@cue] or jsonnet [@jsonnet] are such domain specific languages (DSL), that offer varying support for string interpolation, (strict) typing, functions and validation.

Infrastructure as Code

A prime example for the application of configuration languages are IaaS[7] products. These solutions arise highly complex solutions with regard to resource provision (computing, storage, load balancing, etc.), network setup and scaling. Although the primary interaction with those systems is imperative, maintaining entire applications' or company's environments manually comes with obvious drawbacks.

Changing and undoing changes to existing networks requires intricate knowledge about its topology which in turn has to be meticulously documented as a significant risk for *config drift*. Beyond that, interacting with a system through its imperative interfaces demands qualified skills of specialized engineers.

The concept of "Infrastructure as Code" (*IaC*) serves the DevOps principle of overcoming the need for dedicated teams for *Dev*elvopent and *Op*erations, by allowing to declaratively specify the dependencies, topology and virtual resources. Today various tools with different scopes make it easy to provision complex networks, in a reproducible way. That is setting up the same environment automatically and independently. Optimally, different environments for testing, staging and production can be derived from a common base and changes to configurations are atomic.

As a notable instance, the Nix[@nix] ecosystem even goes as far as enabling declarative system and service configuration using NixOps[@nixops].

To get an idea of how this would look like, lst. 3 shows the configuration for a deployment of the Git based wiki server Gollum[@gollum] behind a nginx reverseproxy on the AWS network. Although targeting AWS, Nix itself is platform-agnostic and NixOps supports different backends through various plugins. Configurations like this are abstractions over many manual steps and the Nix language employed in this example allows for even higher level turing-complete interaction with configurations.

Similarly, tools like Terraform[@terraform], or Chef[@chef] use their own DSLs and integrate with most major cloud providers. The popularity of these products[8], beyond all, highlights the importance of expressive configuration formats and their industry value.

Finally, descriptive data formats for cloud configurations allow mitigating security risks through static analysis. Yet, as recently as spring 2020 and still more than a year later dossiers of Palo Alto Networks' security department Unit 42 show

---

[7]Infrastructure as a Service

[8]https://trends.google.com/trends/explore?date=2012-01-01%202022-01-01&q=%2Fg%2F11g6bg27fp,CloudFormation

[@pa2020H1, ps2021H2] show that a majority of public projects uses insecure configurations. This suggests that techniques[@aws-cloud-formation-security-tests] to automatically check templates are not actively employed, and points out the importance of evaluated configuration languages which can implement passive approaches to security analysis.

Nickel

Gradual typing

Row types

Contracts

Listing 3 Example NixOps deployment to AWS

```
{
  network.description = "Gollum server and reverse proxy";
  defaults =
    { config, pkgs, ... }:
    {
      deployment.targetEnv = "ec2";
      deployment.ec2.accessKeyId = "AKIA...";
      deployment.ec2.keyPair = "...";
      deployment.ec2.privateKey = "...";
      deployment.ec2.securityGroups = pkgs.lib.mkDefault [ "default" ];
      deployment.ec2.region = pkgs.lib.mkDefault "eu-west-1";
      deployment.ec2.instanceType = pkgs.lib.mkDefault "t2.large";
    };

  gollum =
    { config, pkgs, ... }:
    {
      services.gollum = {
        enable = true;
        port = 40273;
      };
      networking.firewall.allowedTCPPorts = [ config.services.gollum.port ];
    };

  reverseproxy =
    { config, pkgs, nodes, ... }:
    let
      gollumPort = nodes.gollum.config.services.gollum.port;
    in
    {

      deployment.ec2.instanceType = "t1.medium";

      services.nginx = {
        enable = true;
        virtualHosts."wiki.example.net".locations."/" = {
          proxyPass = "http://gollum:${toString gollumPort}";
        };
      };
      networking.firewall.allowedTCPPorts = [ 80 ];
    };
}
```