Implementing a language Server for the Nickel Language

Prestudy

Student: Yannik Sander
Supervisor (Tweag): Yann Hamdaoui
Supervisor (KTH): Martin Monperrus
Examiner: Roberto Guanciale

May 27, 2022

Contents

1 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 refactoring 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. The to-date most successful approach is the Language Server Protocol (LSP). The protocol specifies how editors can communicate with language servers which are separate, editor independent implementations of language analyzers. It allows to express a wide variance of language intelligence. LSP servers allow editors to jump to definitions, find usages, decorate elements with additional information inline or when hovering elements, list symbols and much more. The LSP is discussed in more detail in sec. 2.1. These approaches reduce the effort required to bring language intelligence to editors. Instead of rewriting what is essentially the same language extension for every editor, any editor that implements an LSP client can connect to the same language specific server. Since LSP client implementations are independent of the servers, editor communities can focus on developing the best possible and uniform experience which all LSP servers can leverage. 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 developer of the language server is free to choose the optimal implementing language. 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.

The Nickel**nickel?** language referenced in this work is a new Configuration Programming Language developed by Tweag. It is akin to projects like Cue, Dhall, or Nix in that it is an abstraction over pure data description languages such as JSON, YAML or XML. The Nickel project combines the capabilities of the former being a pure functional language based on lambda calculus with JSON data types, gradual typing, higher-order contracts and a record merging operation. As such, it is intended to write safe abstractions over configuration files as employed in Infrastructure as Code for instance.

1.1 Motivation

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

Before the inception of language servers, implementing specialized features for every language of interest was the sole responsibility of the developers of code editors. 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 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[1] from before the introduction of the LSP shows that except Vim and Sublime Text, both exceptional general text editors, the top 10 most popular IDEs were indeed specialized products. Regardless that some IDEs offer support for more languages through (third-party) extensions, developing any sort of language support to N platforms requires the implementation of N integrations. Missing standards, incompatible implementing languages and often proprietary APIs highlight this problem.

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

The Language Server Protocol aims to solve this issue by specifying an 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 expose access to these features through the LSP. This allows developers to focus resources to a single project that is above all unrelated to editor-native APIs for analytics processing code representation and GUI integration. Now only a single implementation of a language server is required,

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

instead of an individual plugin for each editor. Editor maintainers can concentrate on offering the best possible LSP client support independent of the language.

1.1.1 Problem Definition

The problem this thesis will address is the current lack of documentation and evaluation of the applied methods for existing Language Servers.

While most of the implementations of LSP servers are freely available as Open Source Software [ref?], the methodology 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*.

1.2 Research Questions

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[2]:

RQ.1 How to develop a language server for a new language that satisfies its users' needs while being performant enough not to slow them down?

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

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 objective 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.

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

1.3 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 with JSON data types, gradual typing, higher-order contracts and a record merging operation.

1.4 Research Methodology

What are the scientific methods

1.5 Structure of the thesis

2 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.

2.1 Language Server Protocol

The Language Server Protocol is a JSON-RPC based communication specification comprising an LSP client (i.e. editors) and server (also called language server for simplicity). The LSP decouples the development of clients and servers, allowing developers to focus on either side. The LSP defines several capabilities – standardized functions which are remotely executed by the language server. LSP Clients are often implemented as editor extensions facilitating abstraction libraries helping with the interaction with the protocol and editor interface. Language Servers analyse source code sent by the client and may implement any number of capabilities relevant to the language. Since the LSP is both language and editor independent, the same server implementation can serve all LSP compliant clients eliminating the need to redundantly recreate the same code intelligence for every editor.

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[1] 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[2], the protocol allows developers of editors and languages to work independently on the support for new

[1]https://code.visualstudio.com/
[2]https://github.com/microsoft/language-server-protocol/

languages. If supported by both server and client, the LSP now supports more than 24 language features[3] including code completion, code navigation facilities, contextual information such as types or documentation, formatting, and more.

2.1.1 JSON-RPC

the LSP uses JSON-RPC to communicate between a language server and a client. JSON-RPC (v2) [1] is a JSON based lightweight transport independent remote procedure call [2] protocol.

RPC is a paradigm that allows clients to virtually invoke a method at a connected process. The caller sends a well-defined message to a connected process which executes a procedure associated with the request, taking into account any transmitted arguments. Upon invoking a remote procedure, the client suspends the execution of its environment while it awaits an answer of the server, corresponding to a classical local procedure return.

JSON-RPC is a JSON based implementation of RPC. The protocol specifies the format and meaning of the messages exchanged using the protocol, in particular *Request* and *Notification* and *Result* messages. Since the protocol is based on JSON, is possible to be used through any text based communication channel which includes streams or network sockets. In fact, JSON-RPC only specifies a message protocol, leaving the choice of the transport method to the application.

All messages refer to a `method` and a set of `parameters`. Servers are expected to perform the same procedure associated with the requested `method` at any time. In return, clients have to follow the calling conventions for the requested method. Typically, messages are synchronous, i.e., the client awaits a result associated to the method and its parameters before it continues its execution of the calling environment. Hence, requests are marked with an `id` which is included in the result/error message necessarily sent by the server. If the client does not require a response, it can omit the `id` field. The message is then interpreted as a *notification*, which servers cannot respond to.

Part of the JSON-RPC specification is the ability for clients to batch requests and send a list of request or notification objects. In this case, the server should respond with a list of results matching each request, yet is free to process requests concurrently.

2.1.2 Commands and Notifications

The LSP builds on top of the JSON-RPC protocol described in the previous subsection. It defines four sets of commands:

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

The largest group are commands that are relevant in the scope of the currently opened document, e.g. autocompletion, refactoring, inline values and more. In total the LSP defines 33 [3] of such "Language Features." Editors will notify the server about file changes and client side interaction, i.e., opening, closing and renaming files using "Document Synchronization" methods. While most commands are defined in the document scope, i.e., a single actively edited file, the LSP allows clients to communicate changes to files in the opened project. This so called workspace comprises on or more root folders managed by the editor and all files contained in them. "Workspace Features" allow the server to intercept file creation, renaming or deletion to make changes to existing sources in other files. Use cases of these features include updating import paths, changing class names and other automations that are not necessary local to a single file. In addition, the LSP specifies so called "Window Features" which allow the server to control parts of the user interface of the connected editor. For instance, servers may instruct clients to show notifications and progress bars or open files.

2.1.3 Description of Key Methods

the authors of langserver.org [4] identified six "key methods" of the LSP. The methods represent a fundamental set of capabilities, specifically:

1. Code completion Suggest identifiers, methods or values at the cursor position.
2. Hover information Present additional information about an item under the cursor, i.e., types, contracts and documentation.
3. Jump to definition Find and jump to the definition of a local variable or identifier.
4. Find references List all usages of a defined variable.
5. Workspace/Document symbols List all variables in a workspace or document.
6. Diagnostics Analyze source code, i.e., parse and type check and notify the LSP Client if errors arise.

2.1.3.1 Code Completion

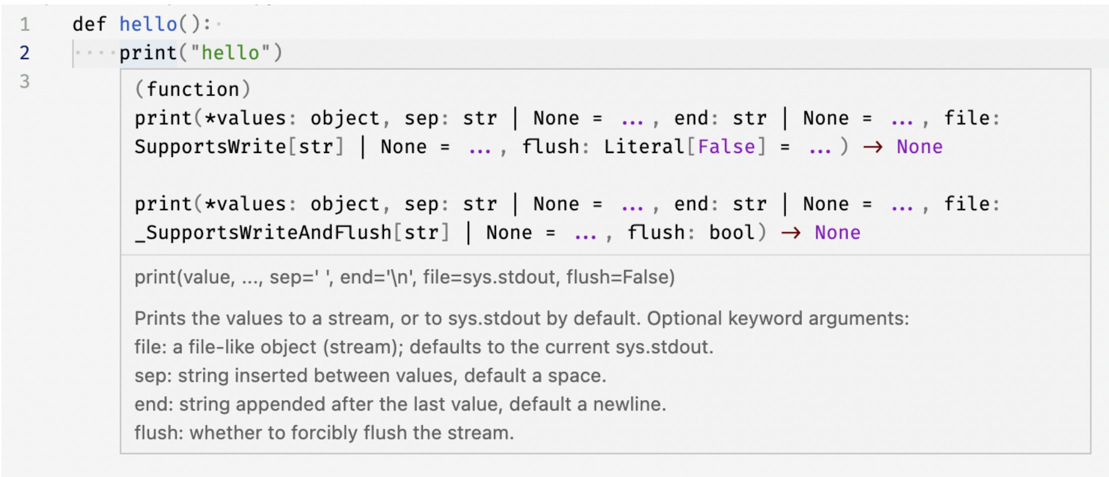
RPC Method: `textDocument/Completion`

This feature allows users to request a suggested identifier of variables or methods, concrete values or larger templates of generated code to be inserted at the position of the cursor. The completion can be invoked manually or upon entering language defined trigger characters, such as `.`, `::` or `->`. The completion request contains the current cursor position, allowing the language server to resolve contextual information based on an internal representation of the document. In the example (fig. **??**) the completion feature suggests related identifiers for the incomplete function call "`print`."

2.1.3.2 Hover

RPC Method: `textDocument/hover`

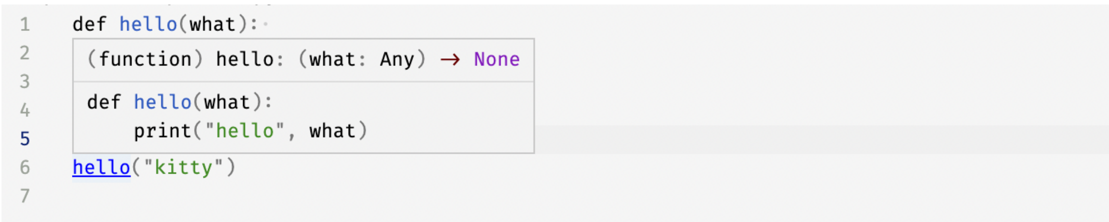
Hover requests are issued by editors when the user rests their mouse cursor on text in an opened document or issues a designated command in editors without mouse support. If the language server has indexed any information corresponding to the position, it can generate a report using plain text and code elements, which are then rendered by the editor. Language servers typically use this to communicate type-signatures or documentation. An example can be seen in fig. **??**.



2.1.3.3 Jump to Definition

RPC Method: `textDocument/definition`

The LSP allows users to navigate their code by the means of symbols by finding the definition of a requested symbol. Symbols can be for instance variable names or function calls. As seen in fig. **??**, editors can use the available information to enrich hover overlays with the hovered element's definition.



2.1.3.4 Find References

RPC Method: ***textDocument/references***

Finding references is the inverse operation to the previously discussed *Jump to Definition* (cf. 2.1.3.3). For a given symbol definition, for example variable, function, function argument or record field the LSP provides all usages of the symbol allowing users to inspect or jump to the referencing code position.

{#fig:lsp-capability-hover caption="Listing of all references to the method"hello". Python language server in Visual Studio Code"}

2.1.3.5 Workspace/Document symbols

RPC Method: `textDocument/workspaceSymbol` *or* `textDocument/documentSymbol`

The symbols capability is defined as both a "Language Feature" and "Workspace Feature" which mainly differ in the scope they represent. The `textDocument/documentSymbol` command lists symbols solely found in the currently opened file, while the `workspace/symbol` takes into account all files in the opened set of folders. The granularity of the listed items is determined by the server and possibly different for either scope. For instance, document symbols could be as detailed as listing any kind of method or property found in the document, while workspace symbols take visibility rules into account which might expose public entities only. Symbols are associated with a span of source code of the symbol itself and its context, for example a function name representing the function body. Moreover, the server can annotate the items with additional attributes such as symbol kinds, tags and even child-references (e.g. for the fields of a record or class).

2.1.3.6 Diagnostics

Diagnostics is the collective term for report statements about the analyzed language of varying severity. This can be parsing or compilation or type-checking n errors, as well as errors and warnings issued by a linting tool.

Unlike the preceding features discussed here, diagnostics are a passive feature, since most often the information stems from external tools being invoked after source code changes. File updates and diagnostics are therefore specified as notifications to avoid blocking the communication.

2.1.4 File Processing

Most language servers handling source code analysis in different ways. The complexity of the language can be a main influence for the choice of the approach. Distinctions appear in the way servers process *file indexes and changes* and how they respond to *requests*.

The LSP supports sending updates in form of diffs of atomic changes and complete transmission of changed files. The former requires incremental parsing and analysis, which are challenging to implement but make processing files much faster upon changes. An incremental approach makes use of an internal representation of the source code that allows efficient updates upon small changes to the source file.

Additionally, to facilitate the parsing, an incremental approach must be able to provide a parser with the right context to correctly parse a changed fragment of code. In practice, most language servers process file changes by re-indexing the entire file, discarding the previous internal state entirely. This is a more approachable method, as it poses less requirements to the architects of the language server. Yet, it is far less performant. Unlike incremental processing (which updates only the affected portion of its internal structure), the smallest changes, including adding or removing lines effect the *reprocessing of the entire file.* While sufficient for small languages and codebases, non-incremental processing quickly becomes a performance bottleneck.

For code analysis LSP implementers have to decide between *lazy* or *greedy* approaches for processing files and answering requests. Dominantly greedy implementations resolve most available information during the indexing of the file. The server can then utilize this model to answer requests using mere lookups. This stands in contrast to lazy approaches where only minimal local information is resolved during the indexing. Requests invoke an ad-hoc resolution the results of which may be memoized for future requests. Lazy resolution is more prevalent in conjunction with incremental indexing, since it further reduces the work associated with file changes. This is essential in complex languages that would otherwise perform a great amount of redundant work.

2.2 Text based Configuration

Configuration languages such as XML[5], JSON[6], or YAML[7] are textual representations of structural data used to configure parameters of a system or application. The objective of such languages is to be machine readable, yet also intuitive enough for humans to write.

2.2.1 Common Configuration Languages

2.2.1.1 JSON

According to the JSON specification [6] the language was designed as a data-interchange format that is easy to read and write both for humans and machines. Since it is a subset of the JavaScript language, its use is particularly straight forward and wide spread in JavaScript based environments such as web applications. But due to its simplicity implementations are available and for all major languages, motivating its use for configuration files.

2.2.1.2 YAML

YAML is another popular language, mainly used for configuration files. According to its goals it should be human readable and independent of the programming language making use of it. It should be efficient to parse and easy to implement while being expressive and extensible.

YAML also features a very flexible syntax which allows for many alternative ways to declare semantically equivalent data. For example boolean expressions can be written as any of the following values:

```
y|Y|yes|Yes|YES|n|N|no|No|NO    true|True|TRUE|false|False|FALSE
on|On|ON|off|Off|OFF
```

Since YAML facilitates indentation as a way to express objects or lists and does not require object keys (and even strings) to be quoted, it is considered easier to write than JSON at the expense of parser complexity. Yet, YAML is compatible with JSON since as subset of its specification defines a JSON equivalent syntax that permits the use of `{..}` and `[..]` to describe objects and lists respectively.

2.2.2 Applications of Configuration Languages

Applications of configuration languages are ubiquitous especially in the vicinity of software development. While XML and JSON are often used by package managers [10], YAML is a popular choice for complex configurations such as CI/CD pipelines [11]–[13] or machine configurations in software defined networks such as Kubernetes[14] and docker compose [15].

Such formats are used due to some significant advantages compared to other binary formats such as databases. Most strikingly, the textual representation allows inspection of a configuration without the need of a separate tool but a text editor. Moreover textual configuration can be version controlled using VCS software like Git which allows changes to be tracked over time. 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.

2.2.3 Configuration Programming Languages

Despite the above-mentioned formats being simple to parse and widely supported [6], 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 [16] or XSD [17].

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 [18] or Vue [19] and python package management using `setuptools`[20]. Yet, the versatility of general purpose languages poses new security risks if used in this context as configurations could now contain malicious code requiring additional verification. Beyond this, not all languages serve as a configuration language, e.g. compiled languages.

However, for particularly complex applications, both advanced features and language independence are desirable. Alternatively to using high level general purpose languages, this demand is addressed by domain specific languages (DSL). Dhall [21], Cue [22] or jsonnet [23] are such domain specific languages, that offer varying support for string interpolation, (strict) typing, functions and validation. Most of these languages are used as a templating system which means a configuration file in a more portable format is generated using an evaluation of the more expressive configuration source. The added expressiveness manifests in the ability to evaluate expression and the availability of imports, variables and functions. These features allow to refactor and simplify repetitive configuration files.

2.2.4 Infrastructure as Code

The shift to an increasing application of IaaS[4] products started the desire for declarative machine configuration in a bid to simplify the setup and reproducibility of such systems. This configuration based setup of infrastructure is commonly summarized as infrastructure as code or IaC. As the name suggests, IaC puts cloud configuration closer to the domain of software development [24].

In principle, IaaS solutions offer great flexibility with regard to resource provision (computing, storage, load balancing, etc.), network setup and scaling of (virtual) servers. However, since the primary interaction with those systems is imperative and based on command line or web interfaces, 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. Undocumented modification pose a significant risk for *config drift* which is particularly difficult to undo imperatively. Beyond that, interacting with a system through its imperative interfaces demands qualified skills of specialized engineers.

The concept of "Infrastructure as Code" (*IaC*) align with the principles of DevOps. IaC tools help to overcome the need for dedicated teams for *Dev*elopment and *Op*erations by allowing to declaratively specify the dependencies, topology and virtual resources. Optimally, different environments for testing, staging and production can be derived from a common base and changes to configurations are atomic. As an additional benefit, configuration code is subject to common software engineering tooling; It can be statically analyzed, refactored and version controlled to ensure reproducibility. A subset of IaC is focused on largely declarative configuration based on configuration files that are interpreted and "converted" into imperative platform dependent instructions.

[4] Infrastructure as a Service

As a notable instance, the Nix[[25];dolstraNixSafePolicyFree2004] ecosystem even goes as far as enabling declarative system and service configuration using NixOps[26].

To get an idea of how this would look like, lst. 2.1 shows the configuration for a deployment of the Git based wiki server Gollum[27] behind a nginx reverse proxy on the AWS network. Although this example targets 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.

Listing 2.1 Example NixOps deployment to AWS

```nix
{
  network.description = "Gollum server and reverse proxy";
  defaults =
    { config, pkgs, ... }:
    {
      # Configuration of a specific deployment implementation
      # here: AWS EC2

      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, ... }:
    {
      # Nix based setup of the gollum server

      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
    {
      # Nix based setup of a nginx reverse proxy

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

      # Instance can override default deployment options
      deployment.ec2.instanceType = "t1.medium";
    };
}
```

Similarly, tools like Terraform[28], or Chef[29] use their own DSLs and integrate with most major cloud providers. The popularity of these products[5], 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 Fall 2021 dossiers of Palo Alto Networks' security department Unit 42 [30] show that a majority of public projects use insecure configurations. This suggests that techniques[31] 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.

2.2.5 Nickel

The Nickel[32] language is a configuration programming language as defined in sec. 2.2.3 with the aims of providing composable, verifiable and validatable configuration files. Nickel implements a pure functional language with JSON-like data types and turing-complete lambda calculus. The language draws inspiration from existing projects such as Cue [22], Dhall [21] and most importantly Nix [25]. However, Nickel sets itself apart from the existing projects by combining and advancing their strengths. The language addresses concerns drawn from the experiences with Nix which employs a sophisticated modules system to provide type-safe, composed (system) configuration files. Nickel implements gradual type annotations, with runtime checked contracts to ensure even complex configurations remain correct. Additionally, considering record merging on a language level facilitates modularization and composition of configurations.

2.2.5.1 Nickel AST

Nickel's syntax tree is a single sum type, i.e., an enumeration of node types. Each enumeration variant may refer to child nodes, representing a branch or hold terminal values in which case it is considered a leaf of the tree. Additionally, tree nodes hold information about their position in the underlying code.

2.2.5.1.1 Basic Elements The primitive values of the Nickel language are closely related to JSON. On the leaf level, Nickel defines `Boolean`, `Number`, `String` and `Null`. In addition to that the language implements native support for `Enum` values which are serialized as plain strings in less expressive formats such as JSON. Each of these are terminal leafs in the syntax tree.

Completing JSON compatibility, `List` and `Record` constructs are present as well. Records on a syntax level are Dictionaries, uniquely associating an identifier with a sub-node.

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

These data types constitute a static subset of Nickel which allows writing JSON compatible expressions as shown in lst. 2.2.

Listing 2.2 Example of a static Nickel expression

```
{
  list = [ 1, "string", null],
  "some key" = "value"
}
```

Beyond these basic elements, Nickel implements variables and functions as well as a special syntax for attaching metadata and recursive records.

2.2.5.1.2 Identifiers The inclusion of variables to the language, implies an understanding of identifiers. Such name bindings can be declared in multiple ways, e.g. `let` bindings, function arguments and records. The usage of a name is always parsed as a single `Var` node wrapping the identifier. Span information of identifiers is preserved by the parser and encoded in the `Ident` type.

Listing 2.3 Let bindings and functions in nickel

```
// simple bindings
let name = <expr> in <expr>
let func = fun arg => <expr> in <expr>

// or with patterns
let name @ { field, with_default = 2 } = <expr> in <expr>
let func = fun arg @ { field, with_default = 2 } =>
  <expr> in
  <expr>
```

2.2.5.1.3 Variable Reference Let bindings in their simplest form merely bind a name to a value expression and expose the name to the inner expression. Hence, the `Let` node contains the binding and links to both implementation and scope subtrees. The binding can be a simple name, a pattern or both by naming the pattern as shown in lst. 2.3.

Listing 2.4 Parsed representation of functions with multiple arguments

```
fun first second => first + second
// ...is parsed as
fun first =>
  fun second => first + second
```

Functions in Nickel are curried lambda expressions. A function with multiple arguments gets broken down into nested single argument functions as seen in lst. 2.4. Function argument name binding therefore looks the same as in `let` bindings.

2.2.5.1.4 Meta Information One key feature of Nickel is its gradual typing system [ref again?], which implies that values can be explicitly typed. Complementing type information, it is possible to annotate values with contracts and additional metadata such as contracts, documentation, default values and merge priority using a special syntax as displayed in lst. 2.5.

Listing 2.5 Example of a static Nickel expression

```
let Contract = {
  foo | Num
      | doc "I am foo",
  hello | Str
        | default = "world"
}
| doc "Just an example Contract"
in
let value | #Contract = { foo = 9, }
in value == { foo = 9, hello = "world", }

> true
```

Internally, the addition of annotations wraps the annotated term in a `MetaValue`, an additional tree node which describes its subtree. The expression shown in lst. 2.6 translates to the AST in fig. 2.1.

Listing 2.6 Example of a typed expression

```
let x: Num = 5 in x
```

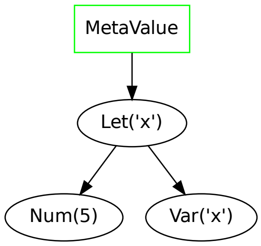


Figure 2.1: AST of typed expression

2.2.5.1.5 Nested Record Access Nickel supports both static and dynamic access to record fields. If the field name is statically known, the access is said to be *static* accordingly. Conversely, if the name requires evaluating a string from an expression the access is called *dynamic*. An example is given in lst. 2.7

Listing 2.7 Examples for static and dynamic record access

```
let r = { foo = 1, "bar space" = 2} in
r.foo // static
r."bar space" // static
let field = "fo" ++ "o" in r."#{field}" // dynamic
```

The destruction of record fields is represented using a special set of AST nodes depending on whether the access is static or dynamic. Static analysis does not evaluate dynamic fields and thus prevents the analysis of any deeper element starting with dynamic access. Static access however can be used to resolve any intermediate reference.

Notably, Nickel represents static access chains in inverse order as unary operations which in turn puts the terminal `Var` node as a leaf in the tree. Figure 2.2 shows the representation of the static access performed in lst. 2.8 with the rest of the tree omitted.

Listing 2.8 Nickel static access

```
let x = {
  y = {
    z = 1,
  }
} in x.y.z
```

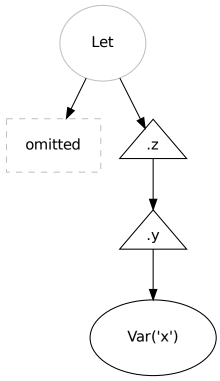


Figure 2.2: AST of typed expression

2.2.5.1.6 Record Shorthand Nickel supports a shorthand syntax to efficiently define nested records similarly to how nested record fields are accessed. As a comparison the example in lst. 2.9 uses the shorthand syntax which resolves to the semantically equivalent record defined in lst. 2.10

Listing 2.9 Nickel record defined using shorthand

```
{
  deeply.nested.record.field = true,
}
```

Listing 2.10 Nickel record defined explicitly

```
{
  deeply = {
    nested = {
      record = {
        field = true,
      }
    }
  }
}
```

Yet, on a syntax level Nickel generates a different representation.

2.2.5.2 Record Merging

Nickel considers record merging as a fundamental operation that combines two records (i.e. JSON objects). Merging is a commutative operation between two records which takes the fields of both records and returns a new record that contains the fields of both operands (cf. lst. 2.11)

Listing 2.11 Merging of two records without shared fields

```
{ enable = true } & { port = 40273 }
>>
{
  enable = true,
  port = 40273
}
```

If both operands contain a nested record referred to under the same name, the merging operation will be applied to these records recursively (cf. lst. 2.12).

Listing 2.12 Merging of two records without shared nested records

```
let enableGollum = {
  service = {
    gollum = {
      enable = true
    }
  }
} in

let setGollumPort = {
  service = {
    gollum = {
      port = 40273
    }
  }
} in

enableGollum & setGollumPort

>>
{
  service = {
    gollum = {
      enable = true,
      port = 40273
    }
  }
}
```

However, if both operands contain a field with the same name that is not a mergeable record, the operation fails since both fields have the same priority making it impossible for Nickel to chose one over the other (cf. lst. 2.13) Specifying one of the fields as a `default` value allows a single override (cf. lst. 2.14). In future versions of Nickel ([33]) it will be possible to specify priorities in even greater detail and provide custom merge functions.

Listing 2.13 Failing merge of two records with common field

```
{ port = 40273 } & { port = 8080 }

>>
error: non mergeable terms
  |
1 | { port = 40273 } & { port = 8080 }
  |          ^^^^^               ^^^^ with this expression
  |          |
  |          cannot merge this expression
```

Listing 2.14 Succeeding merge of two records with default value for common field

```
{ port | default = 40273 } & { port = 8080 }

>>
{ port = 8080 }
```

2.2.5.3 Gradual typing

The typing approach followed by Nickel was introduce by Siek and Taha [34] as a combination of static and dynamic typing. The choice between both type systems is traditionally debated since either approach imposes specific drawbacks. Static typing lacks the flexibility given by fully dynamic systems yet allow to ensure greater correctness by enforcing value domains. While dynamic typing is often used for prototyping, once an application or schema stabilizes, the ability to validate data schemas is usually preferred, often requiring the switch to a different statically typed language. Gradual typing allows introducing statically checked types to a program while allowing other parts of the language to remain untyped and thus interpreted dynamically.

2.2.5.4 Contracts

In addition to a static type-system Nickel integrates a contract system akin what is described in [35]. First introduced by Findler and Felleisen, contracts allow the creation of runtime-checked subtypes. Unlike types, contracts check an annotated value using arbitrary functions that either pass or *blame* the input. Contracts act like assertions that are automatically checked when a value is used or passed to annotated functions.

For instance, a contract could be used to define TCP port numbers, like shown in lst. 2.15.

Listing 2.15 Sample Contract ensuring that a value is a valid TCP port number

```
let Port | doc "A contract for a port number" =
  contracts.from_predicate (
    fun value =>
      builtins.is_num value &&
      value % 1 == 0 &&
      value >= 0 &&
      value <= 65535
  )
in 8080 | #Port
```

Going along gradual typing, contracts pose a convenient alternative to the `newtype` pattern. Instead of requiring values to be wrapped or converted into custom types, contracts are self-contained. As a further advantage, multiple contracts can be applied to the same value as well as integrated into other higher level contracts. An example can be observed in lst. 2.16

Listing 2.16 More advaced use of contracts restricting values to an even smaller domain

```
let Port | doc "A contract for a port number" =
  contracts.from_predicate (
    fun value =>
      builtins.is_num value &&
      value % 1 == 0 &&
      value >= 0 &&
      value <= 65535
  )
in
let UnprivilegedPort = contracts.from_predicate (
  fun value =>
    (value | #Port) >= 1024
  )
in
let Even = fun label value =>
  if value % 2 == 0 then value
  else
    let msg = "not an even value" in
    contracts.blame_with msg label
in

8001 | #UnprivilegedPort
     | #Even
```

Notice how contracts also enable detailed error messages (see lst. 2.17) using custom blame messages. Nickel is able to point to the exact value violating a contract as well as the contract in question.

Listing 2.17 Example error message for failed contract

```
error: Blame error: contract broken by a value [not an even value].
    - :1:1
   |
 1 | #Even
   | ----- expected type
   |
    - repl-input-34:22:1
   |
22 | - 8001 | #UnprivilegedPort
   |   ---- evaluated to this expression
23 | |      | #Even
   | -------------^ applied to this expression

note:
    - repl-input-34:23:8
   |
23 |      | #Even
   |        ^^^^^ bound here
```

3 Related work

The Nickel Language Server follows a history of previous research and development in the domain of modern language tooling and editor integration. Most importantly, it is part of a growing set of LSP integrations. As such, it is important to get a picture of the field of current LSP projects and the approaches leading up to the LSP. This chapter will first introduce and show the shortcomings of classical IDE based language tooling as well as early advances towards editor independent implementations. The following section displays particular instances of different approaches towards the implementation of language servers. Projects that extend and supplement or take inspiration from the LSP, are portrayed in the final section.

3.1 IDE Support

3.1.1 Native and Plugin Systems

Before the invention of the Language Server Protocol, language intelligence used to be provided by an IDE. Yet, the range of officially supported languages remained relatively small[1]. While integration for popular languages was common, top-tier support for less popular ones was all but guaranteed and relied mainly on community efforts. In fact Eclipse[37], IntelliJ[38], and Visual Studio[39], to this day the most popular IDE choices, focus on a narrow subset of languages, historically Java and .NET. Additional languages can be integrated by custom (third-party) plugins or derivatives of the base platform. Due to the technical implications, plugins are generally not compatible between different platforms. Many less popular languages therefore saw redundant implementations of what is essentially the same. For Haskell separate efforts produced an eclipse based IDE[40], as well as independent IntelliJ plugins[42]. Importantly, the implementers of the former reported troubles with the language barrier between Haskell and the Eclipse base written in Java.

The Haskell language is an exceptional example since there is also a native Haskell IDE[43] albeit that it is available only to the MacOS operating system. This showcases the difficulties of language tooling and its provision, since all of these projects are platform dependent and differ in functionality. Moreover, effectively the same tool is developed multiple times wasting resources.

[1] https://www.jetbrains.com/products/

In general, developing language integrations, both as the vendor of an IDE or a third-party plugin developer requires extensive resources. Table 3.1 gives an idea of the efforts required. Since the IntelliJ platform is based on the JVM, its plugin system requires the use of JVM languages[44], making it hard to reuse the code of e.g. a reference compiler or interpreter. The Rust and Haskell integrations for instance contain at best only a fraction of code in their respective language.

Table 3.1: Comparison of the size for different IntelliJ platform plugins

Plugin	lines of code
intellij-haskell	17249 (Java) + 13476 (Scala) **+ 0 (Haskell)**
intellij-rust	229131 (Kotlin) + 3958 (Rust)
intellij-scala	39382 (Java) + 478904 (Scala)
intellij-kotlin	182372 (Java) + 563394 (Kotlin)
intellij-community/python	47720 (C) + 248177 (Java) + 37101 (Kotlin) + 277125 (Python)

Naturally, development efforts at this size tend to gravitate around the most promising solution, stifling the progress on competing platforms. Editor-specific approaches also tend to lock-in programmers into a specific platform for its language support regardless of their personal preference.

3.1.2 Server Client Abstractions

3.1.2.1 Monto

The authors of the Monto project[46] call this the "IDE Portability Problem." They compare the situation with the task of compiling different high level languages to a set of CPU architectures. In Compilers, the answer to that problem was the use of an intermediate representation (IR). A major compiler toolchain making use of this is the LLVM[47]. Compiler frontends for different languages – e.g. Clang[48], Rustc[49], NVCC[50],… – compile input languages into LLVM IR, a low level language with additional capabilities to provide optimization hints but independent of a particular architecture. LLVM performs optimization of the IR and passes it to a compiler backend which in turn generates bytecode for specific architectures e.g. `x86_64`, `MIPS`, `aarch64`, `wasm`, etc. Notably through this mechanism, languages gain support for a range of architectures and profit from existing optimizations developed for the IR.

With Monto, Kreidel et al propose a similar idea for IDE portability. The paper describes the *Monto IR* and how they use a *Message Broker* to receive events from the Editor and dispatch them to *Monto Services*.

The Monto IR is a language-agnostic and editor-independent tree-like model serialized as JSON. Additionally, the IR maintains low level syntax highlighting information (font, color, style, etc.) but leaves the highlighting to the language specific service.

The processing and modification of the source code and IR is performed by *Monto Services*. Services implement specific actions, e.g. parsing, outlining or highlighting. A central broker connects the services with each other and the editor.

Since Monto performs all work on the IR, independent of the editor, and serializes the IR as JSON messages, the language used to implement *Monto Services* can be chosen freely giving even more flexibility.

The Editor extension's responsibility is to act as a source and sink for data. It sends Monto compliant messages to the broker and receives processing results such as (error) reports. The communication is based on the ZeroMQ[51] technology which was chosen because it is lightweight and available in manly languages[46] allowing to make use of existing language tools.

3.1.2.2 Merlin

The Merlin tool[53] is in many ways a more specific version of the idea presented in Monto. Merlin is a language server for the Ocaml language, yet predates the Language Server Protocol.

The authors of Merlin postulate that implementing "tooling support traditionally provided by IDEs" for "niche languages" demands to "share the language-awareness logic" between implementations. As an answer to that, they describe the architecture of Merlin in [53].

Similarly to Monto, Merlin separates editor extensions from language analysis. However, Merlin uses a command line interface instead of message passing for interaction. Editor extensions expose the server functions to the user by integrating with the editor.

Thanks to this architecture, the Merlin developers have been able to focus their efforts on a single project providing intelligence for OCaml source code. The result of this work is a platform independent, performant and fault-tolerant provider of language intelligence. Low level changes to the compiler core have been made to provide incremental parsing, type-checking and analysis. Apart from more efficient handling of source changes, this allows users to query information even about incomplete or incorrect programs.

Since both Merlin and the OCaml compiler are written in OCaml, Merlin is able to reuse large parts of the reference OCaml implementation. This allows Merlin to avoid reimplementing every single feature of the language. Still, incremental parsing and typechecking is not a priority to the compiler which prompted the developers of Merlin to vendor modified versions of the core OCaml components.

While Merlin serves as a single implementation used by all clients, unlike Monto it does not specify a language independent format, or service architecture. In fact, Merlin explicitly specializes in a single language and provides a complete implementation where Monto merely defines the language agnostic interface to implement a server on.

3.2 Language Servers

The LSP project was announced[55] in 2016 to establish a common protocol over which language tooling could communicate with editors. The LSP helps the language intelligence tooling to fully concentrate on source analysis instead of integration with specific editors by building custom GUI elements and being restricted to editors extension interface.

At the time of writing the LSP is available in version '3.16[3]. Microsoft's official website lists 172 implementations of the LSP[56] for an equally impressive number of languages.

An in-depth survey of these is outside the scope of this work. Yet, a few implementations stand out due to their sophisticate architecture, features, popularity or closeness to the presented work.

Since the number of implementations of the LSP is continuously growing, this thesis will present a selected set of notable projects. The presented projects exemplify different approaches with respect to reusing and interacting with the existing language implementation of the targeted language. In particular, the following five approaches are discussed:

1. Three complete implementations that tightly integrate with the implementation level tooling of the respective language: *rust-analyzer[57],* ocaml-lsp/merlin[53] and the *Haskell Language Server[59]
2. A project that indirectly interacts with the language implementation through an interactive programming shell (REPL). *Frege LSP[61]*
3. A Language Server that is completely independent of the target language's runtime. Highlighting how basic LSP support can be implemented even for small languages in terms of userbase and complexity. *rnix-lsp[63]
4. Two projects that facilitate the LSP as an interface to an existing tool via HTTP or command line. *CPAchecker[65] and* CodeCompass[66]
5. An approach to generate language servers from domain specific language specifications[67].

3.2.1 Integrating with the Compiler/Runtime

Today LSP-based solutions serve as the go-to method to implement language analysis tools. Emerging languages in particular take advantage from the flexibility and reach of the LSP. Especially the freedom of choice for the implementing language, is facilitated by multiple languages by integrating modules of the original compiler or runtime into the language server.

3.2.1.1 HLS

For instance the Haskell language server facilitates a plugin system that allows it to integrate with existing tooling projects[2]. Plugins provide capabilities for linting, formatting, documentation and other code actions across multiple compiler versions. This architecture allows writing an LSP in a modular fashion in the targeted language at the expense of requiring HSL to use the same compiler version in use by the IDE and its plugins. This is to ensure API compatibility between plugins and the compiler.

3.2.1.2 Ocaml LSP

Similarly, the Ocaml language service builds on top of existing infrastructure by relying on the Merlin project introduced insec. **??**. Here, the advantages of employing existing language components have been explored even before the LSP.

3.2.1.3 Rust-Analyzer

The rust-analyzer[57] takes an intermediate approach. It does not reuse or modify the existing compiler, but instead implements analysis functionality based on low level components. This way the developers of rust-analyzer have greater freedom to adapt for more advanced features. For instance rust-analyzer implements an analysis optimized parser with support for incremental processing. Due to the complexity of the language, LSP requests are processed lazily, with support for caching to ensure performance. While many parts of the language have been reimplemented with a language-server-context in mind, the analyzer did not however implement detailed linting or the rust-specific borrow checker. For these kinds of analysis, rust-analyzer falls back to calls to the rust build system.

3.2.1.4 Frege LSP

While the previous projects integrated into the compiler pipeline and processed the results separately, other approaches explored the possiblity to shift the entire analysis to existing modules. A good example for this method is given by the Frege language[3].

Frege as introduced in[60] is a JVM based functional language with a Haskell-like syntax. It features lazy evaluation, user-definable operators, type classes and integration with Java.
While previously providing an eclipse plugin[4], the tooling efforts have since been put towards an LSP implementation. The initial development of this language server has

[2]https://hackage.haskell.org/packages/browse?terms=hls
[3]https://github.com/Frege/frege
[4]https://github.com/Frege/eclipse-plugin

been reported on in[61]. The author shows though multiple increments how they utilized the JVM to implement a language server in Java for the (JVM based) Frege language. In the final proof-of-concept, the authors build a minimal language server through the use of Frege's existing REPL and interpreter modules. The file loaded into the REPL environment providing basic syntax and type error reporting. The Frege LSP then translates LSP requests into expressions, evaluates them in the REPL environment and wraps the result in a formal LSP response. Being written in Java, allows the server to make use of other community efforts such as the LSP4J project which provide abstractions over the interaction with LSP clients. Through the use of abstraction like the Frege REPL, servers can focus on the implementation of capabilities only, albeit with the limits set by the interactive environment.

3.2.1.5 Runtime-independent LSP implementations

While many projects do so, language servers do not need to reuse any existing infrastructure of a targeted language at all. Often, language implementations do not expose the required language interfaces (parsing, AST, Types, etc..), or pose various other impediments such as a closed source, licensing, or the absence of LSP abstractions available for the host language.

An instance of this type is the rnix-lsp[63] language server for the Nix[62] programming language. Despite the Nix language being written in C++[25], its language server builds on a custom parser called "rnix[68] in Rust. However, since rnix does not implement an interpreter for nix expressions the rnix based language server is limited to syntactic analysis and changes.

3.2.1.6 Language Server as an Interface to CLI tools

While language servers are commonly used to provide code based analytics and actions such as refactoring, it also proved suitable as a general interface for existing external tools. These programs may provide common LSP features or be used to extend past the LSP.

3.2.1.7 CPAchecker

The work presented by Leimeister in[65] exemplifies how LSP functionality can be provided by external tools. The server can be used to automatically perform software verification in the background using CPAchecker[64]. CPAchecker is a platform for automatic and extensible software verification. The program is written in Java and provides a command line interface to be run locally. Additionally, it is possible to execute resource intensive verification through an HTTP-API on more powerful machines or clusters[70]. The LSP server supports both modes of operation. While it can interface directly with the Java modules provided by the CPAchecker library, it is also able to utilize an HTTP-API provided by a server instance of the verifier.

3.2.1.8 CodeCompass

Similar to the work by Leimeister (c.fsec. 3.2.1.7), in[71] Mészáros et al. present a proof of concept leveraging the LSP to integrate (stand-alone) code comprehension tools with the LSP compliant VSCode editor. Code comprehension tools support the work with complex code bases by "providing various textual information, visualization views and source code metrics on multiple abstraction levels." Pushing the boundaries of LSP use-cases, code comprehension tools do not only analyze specific source code, but also take into account contextual information. One of such tools is CodeCompass[66]. The works of Mészáros yielded a language server that allowed to access the analysis features of CodeCompass in VSCode. In their paper they specifically describe the generation of source code diagrams. Commands issued by the client are processed by a CodeCompass plugin which acts as an LSP server and interacts with CodeCompass through internal APIs.

3.2.2 Language Servers generation for Domain Specific Languages

Bünder and Kuchen[67] highlight the importance of the LSP in the area of Domain Specific Languages (DSL). Compared to general purpose languages, DSLs often targets both technical and non-technical users. While DSL creation workbenches like Xtext[72], Spoofax[73] or MPS[74] allow for the implementation and provision of Eclipse or IntelliJ based DSLs, tooling for these languages is usually tied to the underlying platform. Requiring a specific development platform does not satisfy every user of the language. Developers have their editor of choice, that they don't easily give up on. Non-technical users could easily be overwhelmed by a complex software like Eclipse. For those non-technical users, a light editor would be more adapted, or even one that is directly integrated into their business application. The authors of[67] present how Xtext can generate an LSP server for a custom DSL, providing multi-editor support. The authors especially mention the Monaco Editor[75], a reusable HTML component for code editing using web technologies. It is used in products like VSCode[5], Theia[76] and other web accessible code editors. The Monaco Editor supports the LSP as a client (that is, on the editor side). Such LSP-capable web editors make integrating DSLs directly into web applications easier than ever before.

[5] https://github.com/Frege/eclipse-plugin

3.3 Alternative approaches

3.3.1 LSP Extensions

The LSP defines a large range of commands and capabilities which is continuously being extended by the maintainers of the protocol. Yet, occasionally server developers find themselves in need of functionality not yet present in the protocol. For example the LSP does not provide commands to exchange binary data such as files. Insec. **??** the CodeCompass Language Server was introduced. A stern feature of this server is the ability to generate and show diagrams in SVG format. However, the LSP does not define the concept of *requesting diagrams*. In particular Mészáros et al. describe different shortcomings of the LSP :

1. "LSP doesn't have a feature to place a context menu at an arbitrary spot in the document"

   Context menu entries are implemented by clients based on the agreed upon capabilities of the server. Undefined capabilities cannot be added to the context menu.

   In the case of CodeCompass the developers made up for this by using the code completion feature as an alternative dynamic context menu.

2. "LSP does not support displaying pictures (diagrams)."

   CodeCompass generates diagrams for selected code. Yet, there is no image transfer included with the LSP. Since the LSP is based on JSON-RPC messages, the authors' solution was to define a new command, specifically designed to tackle this non-standard use case.

Missing features of the protocol such as the ones pointed out by Mészáros et al. appear frequently, especially in complex language servers or ones that implement more than basic code processing.

The rust-analyzer defines almost thirty non-standard commands[6], to enable different language specific actions.

Taking the idea of the CodeCompass project further, Rodriguez-Echeverria et al. propose a generic extension of the LSP for graphical modeling[77]. Their approach is based on a generic intermediate representation of graphs which can be generated by language servers and turned into a graphical representation by the client implementation.

Similarly, in[79] the authors describe a method to develop language agnostic LSP extensions. In their work they defined a language server protocol for specification languages (SLSP) which builds on top of the existing LSP. The SLSP defines several extensions that each group the functionality of specific domains. However, unlike other LSP extensions

[6]https://rust-analyzer.github.io/manual.html#features

that are added to facilitate functions of a specific server, SLSP is language agnostic. In effect, the protocol extensions presented by Rask et. al. are reusable across different specification languages, allowing clients to implements a single frontend. Given their successes with the presented work, the authors encourage to build abstract, sharable extensions over language specific ones if possible.

3.3.2 Language Server Index Format

Nowadays, code hosting platforms are an integral part of the developer toolset (GitHub, Sourcegraph, GitLab, Sourceforge, etc.). Those platforms commonly display code simply as text, highlighted at best. LSP-like features would make for a great improvement for code navigation and code reading online. Yet, building these features on language servers would incur redundant and wasteful as a server needed to be started each time a visitor loads a chunk of code. Since the hosted code is most often static and precisely versioned, code analysis could be performed ahead of time, for all files of each version. The LSIF (Language Server index Format) specifies a schema for the output of such ahead of time code analysis. Clients can then provide efficient code intelligence using the pre-computed and standardized index.

The LSIF format encodes a graphical structure which mimics LSP types. Vertices represent higher level concepts such as `documents`, `ranges`, `resultSets` and actual results. The relations between vertices are expressed through the edges.

For instance, hover information as introduced insec. 2.1.3.2 for the interface declaration inlst. 3.1 can be represented using the LSIF. Figure 3.1 visualizes the result (cflst. 3.2). Using this graph an LSIF tool is able to resolve statically determined hover information by performing the following steps.

1. Search for `textDocument/hover` edges.
2. Select the edge that originates at a `range` vertex corresponding to the requested position.
3. Return the target vertex.

Listing 3.1 Exemplary code snippet to showing LSIF formatting

```
export interface ResultSet {
}
```

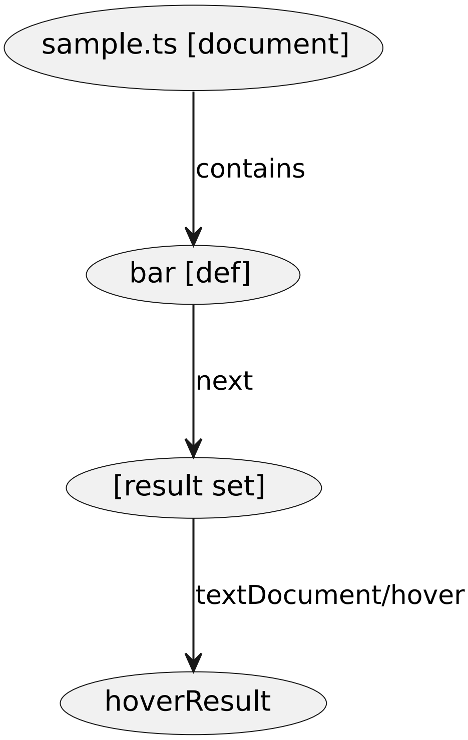


Figure 3.1: LSIF encoded graph for the exemplary code

Listing 3.2 LSIF formated analysis result

```
{ id: 1, type: "vertex", label: "document", uri: "file:///...", languageId: "typescript"
{ id: 2, type: "vertex", label: "resultSet" }
{
   id: 3,
  type: "vertex",
  label: "range",
  start: { line: 0, character: 9},
  end: { line: 0, character: 12 }
}
{ id: 4, type: "edge", label: "contains", outV: 1, inVs: [3] }
{ id: 5, type: "edge", label: "next", outV: 3, inV: 2 }
{
  id: 6,
  type: "vertex",
  label: "hoverResult",
  result: {
    "contents":[
      {"language":"typescript","value":"function bar(): void"},
      ""
    ]
  }
}
{ id: 7, type: "edge", label: "textDocument/hover", outV: 2, inV: 6 }
```

An LSIF report is a mere list of `edge` and `vertex` nodes, which allows it to easily extend and connect more subgraphs, corresponding to more elements and analytics. As a consequence, a subset of LSP capabilities can be provided statically based on the preprocessed LSIF model.

3.3.3 *SP, Abstracting software development processes

Since its introduction the Language Server Protocol has become a standard format to provide language tooling for editing source code. Meanwhile, as hinted insec. 3.3.1, the LSP is not able to fully satisfy every use-case sparking the development of various LSP extensions. Following the success of language servers, similar advances have been made in other parts of the software development process.

For instance, many Java build tools expose software build abstractions through the Build Server Protocol[80], allowing IDEs to integrate more languages more easily by leveraging the same principle as the LSP. The BSP provides abstractions over dependencies, build targets, compilation and running of projects. While the LSP provides `run` or `test` integration for selected languages through Code Lenses, this is not part of the intended responsibilities of the protocol. In contrast, those tasks are explicitly targeted by the BSP.

Next to *writing* software (LSP) and *building/running/testing* software (e.g. BSP), *debugging* presents a third principal task of software development. Similar to the other tasks, most actions and user interfaces related to debugging are common among different languages (stepping in/out of functions, pausing/continuing exection, breakpoints, etc.). Hence, the Debug Adapter Protocol, as maintained by Microsoft and implemented in the VSCode Editor, aims to separate the language specific implementation of debuggers from the UI integration. Following the idea of the LSP, the DAP specifies a communication format between debuggers and editors. Since debuggers are fairly complicated software, the integration of editor communication should not prompt new developments of debuggers. Instead, the DAP assumes a possible intermediate debugger adapter do perform and interface with existing debuggers such as `LLDB`, `GDB`, `node-debug` and others[81].

Following the named protocols, Jeanjean et al. envision a future[82] where all kinds of software tools are developed as protocol based services independent of and shared by different IDEs and Editors. Taking this idea further, they call for a Protocol Specification that allows to describe language protocols on a higher level. Such a protocol, they claim, could enable editor maintainers to implement protocol clients more easily by utilizing automated generation from Language Service Protocol Specifications. Additionally, it could allow different Language Services to interact with and depend on other services.

3.4 References

4 Design and Implementation

This chapter guides through the components of the Nickel Language Server (NLS) as well as the implementation details of the source code analysis and information querying. Aiming for an abstract interface, NLS defines its own data structure underpinning all higher level LSP interactions. Section 4.5 will introduce this linearization data structure and explain how NLS bridges the gap from the explicitly handled Nickel AST. Finally, the implementation of current LSP features is discussed in sec. 4.6.

4.1 Key Objectives

The following points are considered key objectives of this thesis implemented in particular for the Nickel Language Server.

4.1.1 Performance

The usefulness of a language server correlates with its performance. It may cause stutters in the editor, or prompt users to wait for responses when upon issuing LSP commands. Different studies suggest that interruptions are detrimentatl to programmers productivity **interruption-2?**. The more often and longer a task is interrupted the higher the frustration. Hence, as called for in RQ.1 (cf. sec. 1.2), a main criterion for the language server is its performance.

Speaking of language servers there are two tasks that require processing, and could potentially cause interruptions.

Upon source code changes, a language server may reprocess the code to gather general information, and provide diagnostics. Since, for this the LSP uses notifications, and language servers generally run as separate processes, delays in processing may not directly affect the programmer. However, depending on the implementation of the server, multiple changes may queue up preventing the timely response to other requests or delaying diagnostics.

The JSON-RPC protocol underlying the LSP, is a synchronous protocol. Each request requires that the server responded to the previous request before processing can begin. Moreover, the order of requests has to be maintained. Since many requests are issued implicitly by the editor, e.g., hover requests, there is a risk of request queuing which could delay the processing of explicit commands. It is therefore important to provide nearly instantaneous replies to requests.

It is to mention that the LSP defines “long running” requests, that may run in the background. This concept mitigates queuing but can lead to similarly bad user experience as responses appear out of order or late.

4.1.2 Capability

The second objective is to provide an LSP server that offers the most common LSP features as identified by **langserver-org?**. Concretely, these capabilities are:

1. Code completion Suggest identifiers, methods or values at the cursor position.
2. Hover information Present additional information about an item under the cursor, i.e., types, contracts and documentation.
3. Jump to definition Find and jump to the definition of a local variable or identifier.
4. Find references List all usages of a defined variable.
5. Workspace symbols List all variables in a workspace or document.
6. Diagnostics Analyze source code, i.e., parse and type check and notify the LSP Client if errors arise.

For the work on NLS these six capabilities were considered as the goal for a minimal viable product.

4.1.3 Flexibility

The Nickel Language just faced its initial release so changes and additions to the language are inevitable. Since, NLS is expressed as the official tooling solution for the language, it has to be able to keep up with Nickel’s development. Therefore, the architecture needs to be flexible and simple enough to accommodate changes to the language’s structure while remaining the server’s capabilities and requiring little changes to the language core. Likewise, extending the capabilities of the server should be simple enough and designed such future developers are able to pick up the work on NLS.

4.1.4 Generalizability

In the interest of the academic audience and future developers of language servers, this thesis aims to present a reusable solution. The implementation of NLS as examined in this thesis should act as an implementation example that can be applied to other, similar languages. As a result the requirements on the language and its implementation should be minimal. Also, the Language servers should not depend on the implementation of Nickel (e.g. types) too deeply.

4.2 Design Decisions

Section **??** introduced several considerations with respect to the implementation of language servers. Additionally, in sec. **??** presents examples of different servers which guided the decisions made while implementing the NLS. Additionally, in sec. **??** presents examples of different servers which guided the decisions made while implementing the NLS.

4.2.1 Programming language

Rust (**rust?**) was chosen as the implementing language of NLS primarily since Nickel itself is written in Rust. Being written in the same language as the Nickel interpreter allows NLS to integrate existing components for language analysis. This way, changes to the Nickel syntax or code analysis impose minimal adaptation of the Language Server.

In fact, using any other language was never considered since that would have required a separate implementation of integral parts of Nickel, which are actively being developed.

Additionally, Rust has proven itself as a language for LSP Servers. Lastly, Rust has already been employed by multiple LSP servers **lib.rs#language-servers?** which created a rich ecosystem of server abstractions. For instance the largest and most advaced LSP implementation in Rust – the Rust Analyzer **rust-analyzer?** – has contributed many tools such as an LSP server interface **lsp-server-interface?** and a refactoring oriented syntax tree represation **rowan?**. Additionally, lots of smaller languages **mojom?** implement Language Servers in Rust. Rust appears to be a viable choice even for languages that are not originally implemented in Rust, such as Nix **rninx-lsp?**.

In Rust `traits` **traits?** are the fundamental concept used to abstract methods from the underlying data.
Traits are definitions of shared behavior. Similar to interfaces in other languages, a trait defines a set of methods. One implements a trait for a certain type, by defining the behavior in the context of the type. Rust's support for generics**generics?** allows constraining arguments and structure fields to implementors of a certain trait allowing to abstract concrete behavior from its interface.

Rust also excels due to its various safety features and performance, for the following reasons. Safety comes in form of *memory* safety, which is enforced by Rust's ownership model**rust-ownership-model?** and explicit memory handling. The developer in turn needs to be aware of the implications of stack or heap located variables and their size in memory. A different kind of safety is *type* safety which is an implication of Rust's strong type system and `trait` based generics. Type-safe languages such as Rust enforce explicit usage of data types for variables and function definitions. Type annotations ensure that methods and fields can be accessed as part of the compilation saving users from passing incompatible data to functions. This eliminating a common runtime failures as seen in dynamic languages like Python or JavaScript. Finally, as Rust leverages the

LLVM infrastructure and requires no runtime, its performance rivals the traditional C languages.

4.2.2 File processing

Earlier two differnet file processing models were discussed in sec. **??**, incremental and complete processing.

LSP implementations may employ so-called incremental parsing, which allows updating only the relevant parts of its source code model upon small changes in the source. However, an incremental LSP is not trivial to implement, which is why it is mainly found in more complex servers such as the Rust Analyzer **rust-analyzer?** or the OCaml Language Server **merlin?**.

Implementing an incremental LSP server for Nickel would be impractical. NLS would not be able to leverage existing components from the non-incremental Nickel implementation (most notably, the parser). Parts of the nickel runtime, such as the type checker, would need to be adapted or even reimplemented to work incrementally too. Considering the scope of this thesis, the presented approach performs a complete analysis on every update to the source file. The typical size of Nickel projects is assumed to remain small for quite some time, giving reasonable performance in practice. Incremental parsing, type-checking and analysis can still be implemented as a second step in the future after gathering more usage data once nickel and the NLS enjoy greater adoption.

4.2.3 Code Analysis

Code analysis approaches as introduced in sec. **??** can have both *lazy* and *eager* qualities. Lazy solutions are generally more compatible with an incremental processing model, since these aim to minimizing the change induced computation. NLS prioritizes to optimize for efficient queries to a pre-processed data model. Similar to the file processing argument in sec. **??**, it is assumed that Nickel project's size allows for efficient enough eager analysis prioritizing a more straight forward implementation over optimized performance.

An example workflow of both lazy and eager processing is examplified in the sequence diagrams fig. 4.1 and fig. 4.2 respectively. As mentioned in the previous paragraph, it is assumed that the performance gains of direct lookup after an "Ahead of time analysis," outperform the lazy analysis in terms of responsiveness. At the same time the initial analysis is expected to complete in reasonably short time for typical Nickel workflows.

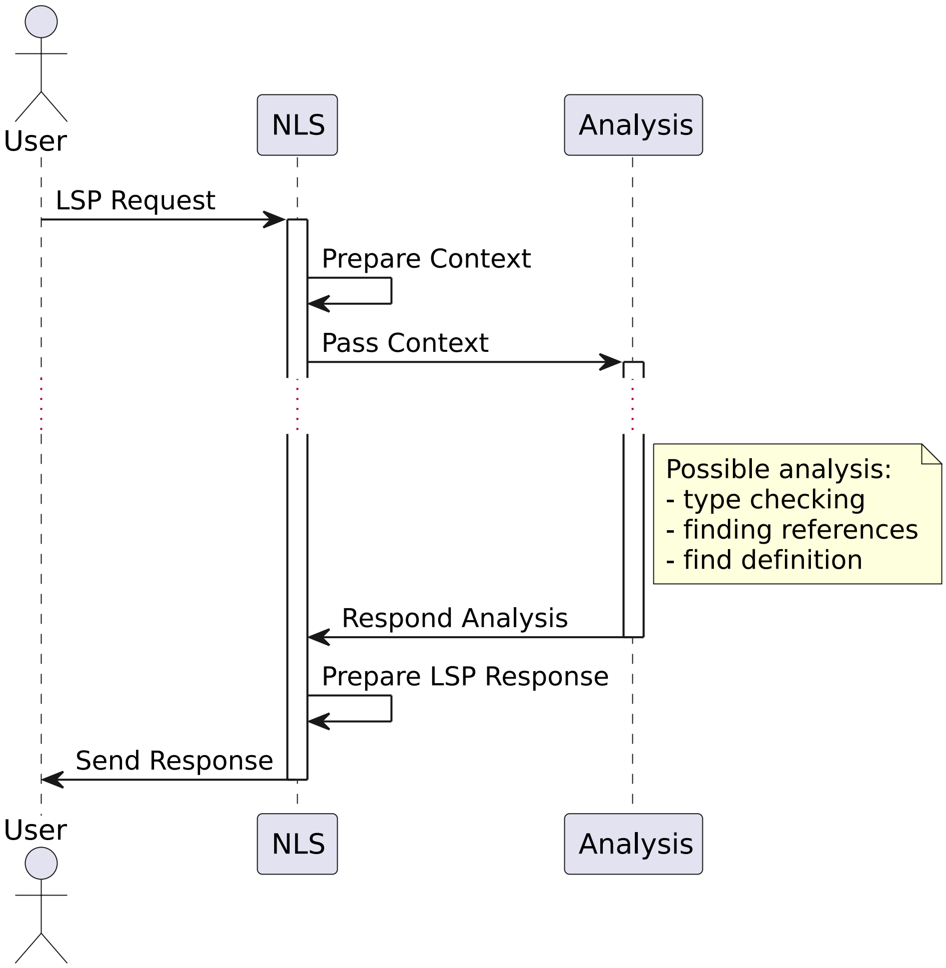


Figure 4.1: Sequence diagram depicting lazy handling of LSP requests.

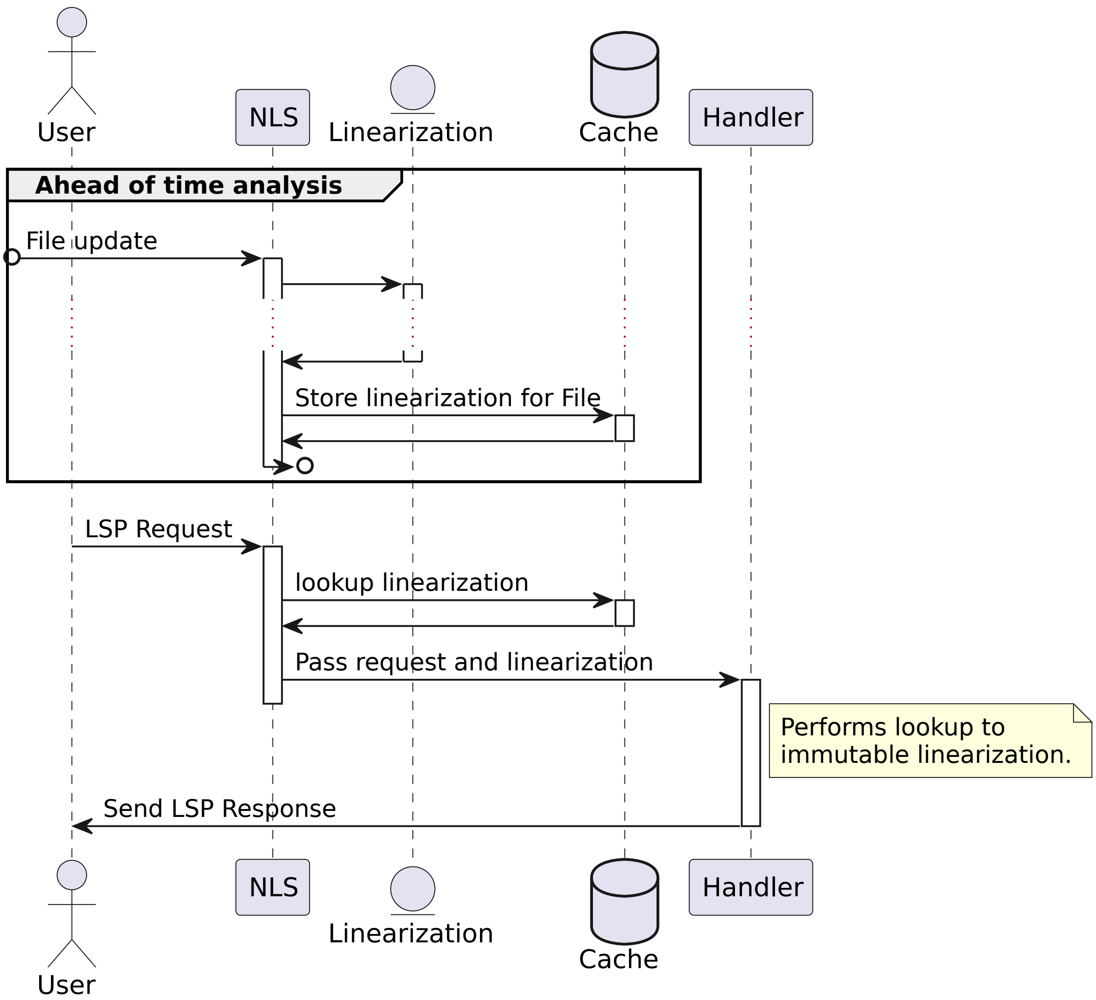


Figure 4.2: Sequence diagram depicting eager analysis and handling of LSP requests.

4.3 High-Level Architecture

This section describes the high-level architecture of NLS. The entity diagram depicted in fig. 4.3 shows the main elements at play.

NLS needs to meet the flexibility and generalizability requirements as discussed in sec. 4.1.3. In short three main considerations have to be satisfied:

1. To keep up with the frequent changes to the Nickel language and ensure compatibility at minimal cost, NLS needs to *integrate critical functions* of Nickel's runtime
2. Adaptions to Nickel to accommodate the language server should be minimal not obstruct its development and *maintain performance of the runtime*.
3. To allow the adoption in other languages, the core language server should be separable from the nickel specifics.

The architecture of NLS reflects these goals, using conceptional groups. The core group labeled "Language Server," contains modules concerning both the source code analysis and LSP interaction. The analysis is base on an internal representation of source code called `Linearization` which can be in one of two states, namely `Building` or `Completed`. Either state manages an array of items (`LinearizationItems`) that are derived from AST nodes as well as various metadata facilitating the actions related to the state. The `LinearizationItem` is an abstract representation of code units represented by AST nodes or generated to support an AST derived item. Items associate a certain span with its type, metadata, scope and a unique id, making it referable to. Additionally, `LinearizationItem`s are assigned a `TermKind` which distinguishes different functions of the item in the context of the linearization. The building of the linearization is abstracted in the `Linearizer` trait. Implementors of this trait convert AST nodes to linearization items and append said items to a shared linearization in the building state. Finally, linearizers define how to post-process and complete the linearization. The full linearization is described in detail in sec. 4.5. The LSP capabilities are implemented as independent functions satisfying the same interface, accepting request parameters, and a reference to the completed linearization. A reference to the server finally allows the handlers to send responses to LSP clients. To facilitate most functions of the linearization and the LSP handlers, the language-server abstraction also defines a range of support types.

Unlike the abstract language server module, the NLS module defines language specific implementations. In particular, it implements the `Linearizer` trait through `AnalysisHost` which is referred to in the following of this document simply as the "linearizer." The linearizer abstracts Nickel's AST nodes into linearization items. Since the linearizer implementation is the only interface between Nickel and NLS, changes to the language that affect the AST require changes to this module only. Representing the main binary, the Server module integrates the Nickel parser and type checker to perform deeper analysis based on an AST representation and to provide diagnostics to the client. Moreover, the integration of Nickel's original modules avoids the need to rewrite these functions which allows NLS to profit from improvements with minimal adaption. The analysis results are cached internally and used by the individual capability handlers to answer LSP requests.

The Nickel module contains, apart from the parsing and type checking functions, a group of types related to AST nodes and the linearization process. While these types currently appear throughout the entire architecture, in the future the language server library will use different abstractions to remove this dependency. Section 6.2 lays out a more detailed plan how this will be achieved.

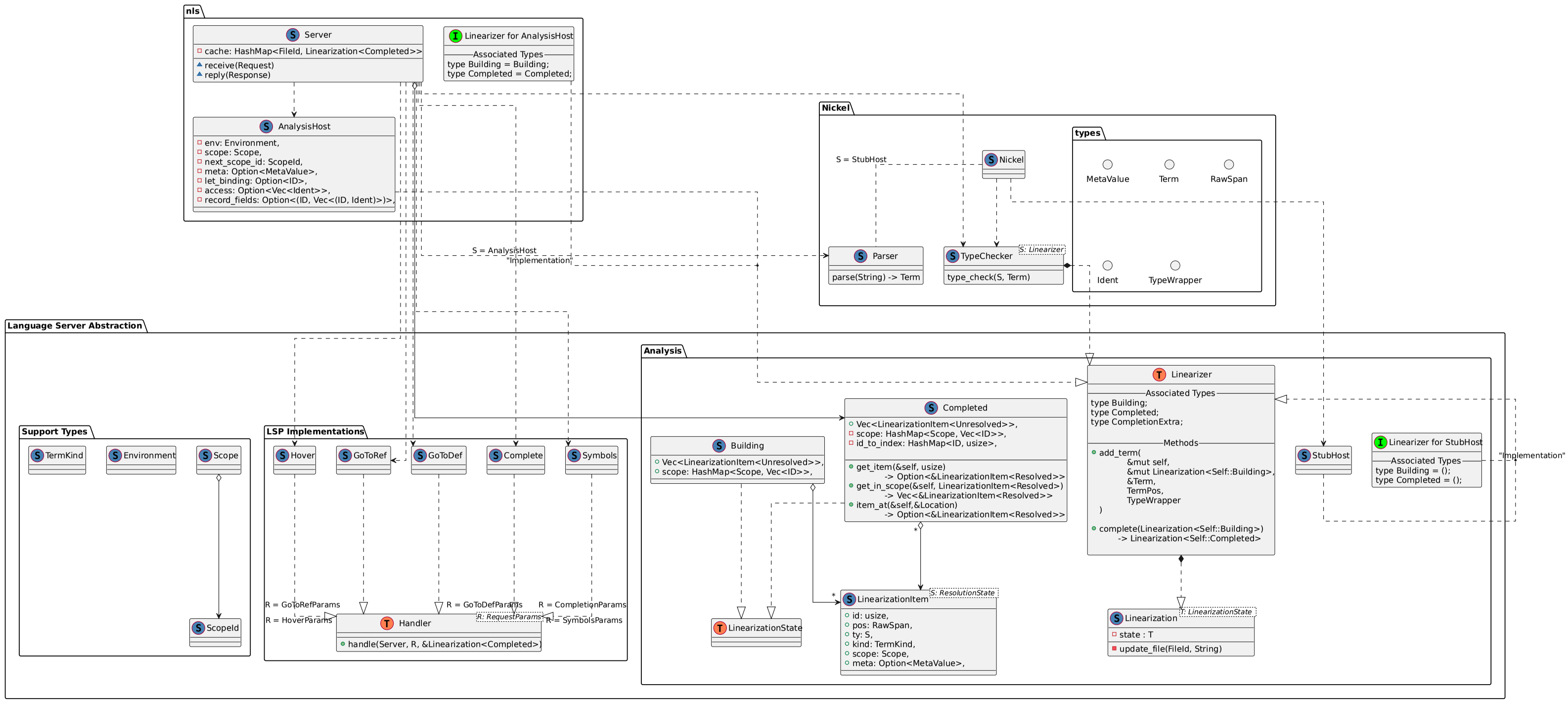


Figure 4.3: Entity Diagram showing the architecture of NLS, explicit dependency arrows omitted for legibility.

4.4 Illustrative example

The example lst. 4.1 shows an illustrative high level configuration of a server. Using Nickel, this file would be used to define the schema of the configuration format of another program. Evaluation and validation is done in the context of Nickel, after which the evaluated structure is translated into a more common (but less expressive) format such as YAML or JSON. Here, the schema for a configuration of a Kubernetes-like **kubernetes?** tool is defined using contracts, making exemplary use of variables and functions. Specifically, it describes a way to provision named containers. The user is able to specify container images and opened ports, as well as define metadata of the deployment. The configuration is constrained by the `NobernetesConfig` contract. The contract in turn defines the required fields and field types. Notably, the fields `containers` and `replicas` are further constrained by individual contracts. The `Port` contract is a logical contract that ensures the value is in the range of valid port numbers. The example also shows different ways of declaring types (i.e. constraining record value types), string interpolation, as well as the usage of let bindings with standard types. Throughout this chapter, different sections about the NSL implementation will refer back to this example.

Listing 4.1 Nickel example with most features shown

```
let Port | doc "A contract for a port number" =
  contracts.from_predicate (fun value =>
    builtins.is_num value &&
    value % 1 == 0 &&
    value >= 0 &&
    value <= 65535) in

let Container = {
  image | Str,
  ports | List Port,
} in

let NobernetesConfig = {
  apiVersion | Str,
  metadata.name | Str,
  replicas | nums.PosNat
           | doc "The number of replicas"
           | default = 1,
  containers | { _ : Container },

} in

let name_ = "myApp" in

let metadata_ = {
    name = name_,
} in

let webContainer = fun image => {
  image = image,
  ports = [ 80, 443 ],
} in

let image = "k8s.gcr.io/%{name_}" in

{
  apiVersion = "1.1.0",
  metadata = metadata_,
  replicas = 3,
  containers = {
    "main container" = webContainer image
  }
} | NobernetesConfig
```

4.5 Linearization

The focus of the NLS as presented in this work is to implement a foundational set of LSP features as described in sec. 4.1.2. In order to process these capabilities efficiently as per sec. 4.1.1, NLS needs to store more information than what is originally present in a Nickel AST (cf. sec. 2.2.5.1), such as information about references. Although this can be deduced from the AST lazily, working with Nickel's tree representation is inefficient, as it is not optimized for random access and search operations. Therefore, as hinted in sec. 4.2.3, optimization is directed to efficient lookup from a pre-processed report. Since most LSP commands refer to code positions, the intermediate structure must allow efficient lookup of analysis results based on positions.

To that end NLS introduces an auxiliary data structure, the so-called linearization. The linearization is a linear representation of the program and consists of linearization items. It is derived node by node, from the program's AST by the means of a recursive tree traversal. The transfer process generates a set of linearization items for every node. The kind of the items as well as any additional type information and metadata are determined by the state of the linearization, and the implementation of the process, also called linearizer. Transferring AST nodes into an intermediate structure has the additional advantage of establishing a boundary between the language dependent and generic part of the language server, since linearization items could be implemented entirely language independent. The transfer process is described in greater detail in sec. 4.5.2.

The linearization can be in the following two different general states that align with the two phases of its life cycle. While NLS processes the AST, it is considered to be in a building state. After the AST is fully transferred, the linearization enters the second, phase in which it is referred to as completed and used by the server to facilitate answering LSP requests. The two states are syntactically separate and implementation dependent through the use of different types and the generic interface that allows the independent implementations of the linearizer. Since different types represent the two states, the building state is explicitly transformed into a completed type allowing for additional post-processing (cf. sec. 4.5.3). To fully support all actions implemented by the server, the completed linearization provides several methods to access specific items efficiently. The implementation of these methods is explained in sec. 4.5.4.

4.5.1 States

At its core the linearization in either state is represented by an array of `LinearizationItem`s which are derived from AST nodes during the linearization process. However, the exact structure of that array differs as an effect of the post-processing.

`LinearizationItem`s maintain the position of their AST counterpart as well as its type. Unlike in the AST ([sec:meta-information]), *metadata* is directly associated with the element. Further deviating from the AST representation, both the *type* of the node and its references to other items are encoded explicitly in the `LinearizationType`. The references form an implicit usage graph on top of the linear structure. It distinguishes between declarations (`let` bindings, function parameters, records) and variable usages. Any other kind of structure, for instance, primitive values (Strings, numbers, boolean, enumerations), is recorded as `Structure`.

To separate the phases of the elaboration of the linearization in a type-safe way, the implementation is based on type-states**typestate?**. Type-states were chosen over an enumeration based approach for the additional flexibility they provide to build a generic interface. First, type-states allow to implement separate utility methods for either state and enforce specific states on the type level. Second the `Linearization` struct provides a common context for all states like an enumeration, yet statically determining the Variant. Additionally, the `Linearizer` trait can be implemented for arbitrary `LinearizationState`s. This allows other LSP implementations to base on the same core while providing, for example, more information during the building phase. The unit type `()` is a so called "zero sized type" [**zero-sized-type?**), it represents the absence of a value. NLS provides a `Linearizer` implementation based on unit types and empty method definitions. As a result, the memory footprint of this linearizer is effectively zero and most method calls will be removed as part of compile time optimizations.

NLS defines two type-state variants according to the two phases of the linearization: `Building` and `Completed`.

building phase: A linearization in the `Building` state is a linearization under construction. It is a list of `LinearizationItem`s of unresolved type, appended as they are created during a depth-first traversal of the AST.
During this phase, the `id` affected to a new item is always equal to its index in the array.
The Building state also records the definitions in scope of each item in a separate mapping.

post-processing phase: Once fully built, a Building instance is post-processed to get a `Completed` linearization.
Although fundamentally still represented by an array, a completed linearization is optimized for search by positions (in the source file) thanks to sorting and the use of an auxiliary map from `id`s to the new index of items.
Additionally, missing edges in the usage graph have been created and the types of items are fully resolved in a completed linearization.

Type definitions of the `Linearization` as well as its type-states `Building` and `Completed` are listed in lsts. 4.2, 4.3, 4.4. As hinted above, the `Linearization` struct acts as the overarching context only. Therefore it is similar to an enumeration where the concrete variants are unknown but statically determined at compile time. The `LinearizationState`s can be implemented according to the needs of the Linearizer implementation of the LSP server built on top of the core module. Note that only the former is defined as part of the Nickel libraries, the latter are specific implementations for NLS.

Listing 4.2 Definition of Linearization structure

```
pub trait LinearizationState {}

pub struct Linearization<S: LinearizationState> {
    pub state: S,
}
```

Listing 4.3 Type Definition of Building state

```
pub struct Building {
    pub linearization: Vec<LinearizationItem<Unresolved>>,
    pub scope: HashMap<Vec<ScopeId>, Vec<ID>>,
}
impl LinearizationState for Building {}
```

Listing 4.4 Type Definition of Completed state

```
pub struct Completed {
    pub linearization: Vec<LinearizationItem<Resolved>>,
    scope: HashMap<Vec<ScopeId>, Vec<ID>>,
    id_to_index: HashMap<ID, usize>,
}
impl LinearizationState for Completed {}
```

4.5.2 Transfer from AST

The NLS project aims to present a transferable architecture that can be adapted for future languages as elaborated in sec. 4.1.4. Consequently, NLS faces the challenge of satisfying multiple goals

1. The core of the server should be language independent.
2. Language dependent features should serve the core abstractions.
3. To keep up with Nickel's rapid development ensuring compatibility at minimal cost, critical functions should integrate with the Nickel language implementation.

4. Adaptions to Nickel should be minimal not obstruct its development and runtime performance.

To accommodate these goals NLS comprises three different parts as shown in fig. **??**.

The `Linearizer` trait acts as an interface between Nickel and the language server. NLS implements a `Linearizer` specialized to Nickel which registers AST nodes and builds a final linearization. Nickel's type checking implementation was adapted to pass AST nodes to the `Linearizer`. Modifications to Nickel are minimal, comprising only few additional function calls and a slightly extended argument list. A stub implementation of the `Linearizer` trait is used during normal operation of the interpreter. Since most methods of this implementation are `no-ops`, the compiler is expected to be able to remove most `Linearizer` related method calls in optimized release builds. This promises minimal runtime impact incurred by the integration of lsp APIs.

4.5.2.1 Usage Graph

At the core the linearization is a simple *linear* structure. Yet, it represents relationships of nodes on a structural level as a tree-like structure. Taking into account variable usage information adds back-edges to the original AST, yielding a graph structure. Both kinds of edges have to be encoded with the elements in the list. Alas, items have to be referred to using `id`s since the index of items cannot be relied on (such as in e.g., a binary heap), because the array is reordered to optimize access by source position.

There are two groups of vertices in such a graph. **Declarations** are nodes that introduce an identifier, and can be referred to by a set of nodes. Referral is represented by **Usage** nodes.

During the linearization process this graphical model is embedded into the items of the linearization. Hence, each `LinearizationItem` is associated with a kind representing the item's role in the graph (see: lst. 4.5).

Listing 4.5 Definition of a linearization items TermKind

```rust
pub enum TermKind {
    Declaration(Ident, Vec<ID>, ValueState),
    Record(HashMap<Ident, ID>),
    RecordField {
        ident: Ident,
        record: ID,
        usages: Vec<ID>,
        value: ValueState,
    },

    Usage(UsageState),

    Structure,
}

pub enum UsageState {
    Unbound,
    Resolved(ID),
    Deferred { parent: ID, child: Ident },
}

pub enum ValueState {
    Unknown,
    Known(ID),
}
```

Variable bindings and function arguments are linearized using the `Declaration` variant which holds

- the bound identifier
- a list of `ID`s corresponding to its `Usage`s.
- its assigned value

Records remain similar to their AST representation. The `Record` variant simply maps the record's field names to the linked `RecordField`

Record fields are represented as `RecordField` kinds and store:

- the same data as for identifiers (and, in particular, tracks its usages)
- a link to the parent `Record`
- a link to the value of the field

Variable usages can be in three different states.

1. `Usage`s that can not (yet) be mapped to a declaration are tagged `Unbound`
2. A `Resolved` usage introduces a back-link to the complementary `Declaration`
3. For record destructuring resolution of the name might need to be `Deferred` to the post-processing as discussed in sec. **??**.

Other nodes of the AST that do not participate in the usage graph, are linearized as `Structure` – A wildcard variant with no associated data.

4.5.2.2 Scopes

The Nickel language implements lexical scopes with name shadowing.

1. A name can only be referred to after it has been defined
2. A name can be redefined locally

An AST supports this concept due to its hierarchical structure. A variable reference always refers to the closest parent node defining the name and scopes are naturally separated using branching. Each branch of a node represents a sub-scope of its parent, i.e., new declarations made in one branch are not visible in the other.

When eliminating the tree structure, scopes have to be maintained. This is to provide LSP capabilities such as auto-completion sec. **??** of identifiers and list symbol names sec. 4.6.3.4, which require the item's scope as context. Since the bare linear data structure cannot be used to deduce a scope, related metadata has to be tracked separately. The language server maintains a register for identifiers defined in every scope. This register allows NLS to resolve possible completion targets as detailed in sec. 4.5.4.3.

The globally tracked scope metadata maps `ScopeId`s to a list of identifiers defined in the scope. An instance of the linearizer is valid for a single scope and hence corresponds to a unique `ScopeId`. Every item generated by the same linearizer is associated with the `ScopeId` of the instance. A scope branch during the traversal of the AST is indicated through the `Linearizer::scope()` method. The `Linearizer::scope()` method creates a new linearizer instance with a new `ScopeId`. A `ScopeId` in turn is a "scope path," a list of path elements where the prefix is equal to the parent scope's `ScopeId`. Listing 4.6 shows the scopes for a simple expression in Nickel explictly.

Listing 4.6 Explicit display of Nickel scopes

```
---------------------------------------------+ /1
                                             |
let record                                   |
  = { ------------------------------+ /1/1   |
                                    |        |
    key1 = "value", -------- /1/1/1  |        |
    key2 = 123, ------------ /1/1/2  |        |
                                    |        |
  } --------------------------------+        |
  in record -------------------------- /1/2  |
                                             |
---------------------------------------------+
```

Additionally, to keep track of the variables in scope, and iteratively build a usage graph, NLS keeps track of the latest definition of each variable name and which `Declaration` node it refers to.

4.5.2.3 Linearizer

The heart of the linearization is the `Linearizer` trait as defined in lst. 4.7. The `Linearizer` lives in parallel to the `Linearization`. Its methods modify a shared reference to a `Building Linearization`.

Listing 4.7 Interface of linearizer trait

```
pub trait Linearizer {
    type Building: LinearizationState + Default;
    type Completed: LinearizationState + Default;
    type CompletionExtra;

    fn add_term(
        &mut self,
        lin: &mut Linearization<Self::Building>,
        term: &Term,
        pos: TermPos,
        ty: TypeWrapper,
    )

    fn retype_ident(
        &mut self,
        lin: &mut Linearization<Self::Building>,
        ident: &Ident,
        new_type: TypeWrapper,
    )

    fn complete(
        self,
        _lin: Linearization<Self::Building>,
        _extra: Self::CompletionExtra,
    ) -> Linearization<Self::Completed>
    where
        Self: Sized,

    fn scope(&mut self) -> Self;
}
```

`Linearizer::add_term` is used to record a new term, i.e. AST node.
Its responsibility is to combine context information stored in the `Linearizer` and concrete information about a node to extend the `Linearization` by appropriate items.

`Linearizer::retype_ident` is used to update the type information of an identifier.
The reason this method exists is that not all variable definitions have a corresponding AST node but may be part of another node. This is the case with records; Field *names* are not linearized separately but as part of the record. Thus, their type is not known to the linearizer and has to be added explicitly.

`Linearizer::complete` implements the post-processing necessary to turn a final

`Building` linearization into a `Completed` one.
Note that the post-processing might depend on additional data.

`Linearizer::scope` returns a new `Linearizer` to be used for a sub-scope of the current one.
Multiple calls to this method yield unique instances, each with their own scope. It is the caller's responsibility to call this method whenever a new scope is entered traversing the AST.
The recursive traversal of an AST implies that scopes are correctly backtracked.

While data stored in the `Linearizer::Building` state will be accessible at any point in the linearization process, the `Linearizer` is considered to be *scope safe*. No instance data is propagated back to the outer scopes `Linearizer`. Neither have `Linearizer`s of sibling scopes access to each other's data. Yet, the `scope` method can be implemented to pass arbitrary state down to the scoped instance. The scope safe storage of the `Linearizer` implemented by NLS, as seen in lst. **??**, stores the scope aware register and scope related data. Additionally, it contains fields to allow the linearization of records and record destructuring, as well as metadata (sec. 4.5.2.4.3).

```rust
pub struct AnalysisHost {
    env: Environment,
    scope: Scope,
    next_scope_id: ScopeId,
    meta: Option<MetaValue>,
    /// Indexing a record will store a reference to the record as
    /// well as its fields.
    /// [Self::Scope] will produce a host with a single **`pop`ed**
    /// Ident. As fields are typechecked in the same order, each
    /// in their own scope immediately after the record, which
    /// gives the corresponding record field _term_ to the ident
    /// useable to construct a vale declaration.
    record_fields: Option<(usize, Vec<(usize, Ident)>)>,
    /// Accesses to nested records are recorded recursively.
    /// ```
    /// outer.middle.inner -> inner(middle(outer))
    /// ```
    /// To resolve those inner fields, accessors (`inner`, `middle`)
    /// are recorded first until a variable (`outer`). is found.
    /// Then, access to all nested records are resolved at once.
    access: Option<Vec<Ident>>,
}
```

4.5.2.4 Linearization Process

From the perspective of the language server, building a linearization is a completely passive process. For each analysis NLS initializes an empty linearization in the `Building` state. This linearization is then passed into Nickel's type-checker along a `Linearizer` instance.

Type checking in Nickel is implemented as a complete recursive depth-first preorder traversal of the AST. As such it could easily be adapted to interact with a `Linearizer` since every node is visited and both type and scope information is available without the additional cost of a separate traversal. Moreover, type checking proved optimal to interact with traversal as most transformations of the AST happen afterwards.

While the type checking algorithm is complex only a fraction is of importance for the linearization. Reducing the type checking function to what is relevant to the linearization process yields lst. 4.8. Essentially, every term is unconditionally registered by the linearization. This is enough to handle a large subset of Nickel. In fact, only records, let bindings and function definitions require additional change to enrich identifiers they define with type information.

Listing 4.8 Abstract type checking function

```
fn type_check_<L: Linearizer>(
    lin: &mut Linearization<L::Building>,
    mut linearizer: L,
    rt: &RichTerm,
    ty: TypeWrapper,
    /* omitted */
) -> Result<(), TypecheckError> {
    let RichTerm { term: t, pos } = rt;

    // 1. record a node
    linearizer.add_term(lin, t, *pos, ty.clone());

    // handling of each term variant
    // recursively calling `type_check_`
    //
    // 2. retype identifiers if needed
    match t.as_ref() {
      Term::RecRecord(stat_map, ..) => {
        for (id, rt) in stat_map {
          let tyw = binding_type(/* omitted */);
          linearizer.retype_ident(lin, id, tyw);
        }
      }
      Term::Fun(ident, _) |
      Term::FunPattern(Some(ident), _)=> {
        let src = state.table.fresh_unif_var();
        linearizer.retype_ident(lin, ident, src.clone());
      }
      Term::Let(ident, ..) |
      Term::LetPattern(Some(ident), ..)=> {
        let ty_let = binding_type(/* omitted */);
        linearizer.retype_ident(lin, ident, ty_let.clone());
      }
      _ => { /* omitted */ }
    }
```

While registering a node, NLS distinguishes 4 kinds of nodes. These are *metadata*, *usage graph* related nodes, i.e. declarations and usages, *static access* of nested record fields, and *general elements* which is every node that does not fall into one of the prior categories.

Listing 4.9 Exemplary nickel expressions

```
// atoms

1
true
null

// binary operations
42 * 3
[ 1, 2, 3 ] @ [ 4, 5]

// if-then-else
if true then "TRUE :)" else "false :("

// string iterpolation
"%{ "hello" } %{ "world" }!"
```

4.5.2.4.1 Structures In the most common case of general elements, the node is simply registered as a `LinearizationItem` of kind `Structure`. This applies for all simple expressions like those exemplified in lst. 4.9

4.5.2.4.2 Declarations In case of `let` bindings or function arguments name binding is equally simple. As discussed in sec. **??** the `let` node may contain both a name and patterns. For either the linearizer generates `Declaration` items and updates its name register. However, type information is available for name bindings only, meaning pattern matches remain untyped.

The same process applies for argument names in function declarations. Due to argument currying **currying?**, NLS linearizes only a single argument/pattern at a time.

Listing 4.10 A record in Nickel

```
{
  apiVersion = "1.1.0",
  metadata = metadata_,
  replicas = 3,
  containers = {
    "main container" = webContainer image
  }
}
```

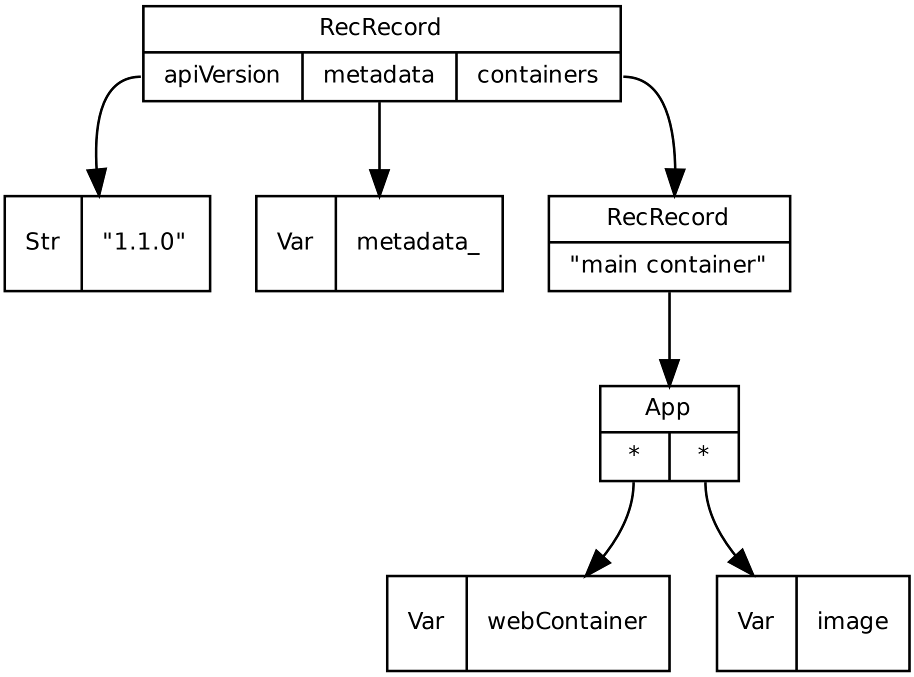


Figure 4.4: AST representation of a record

4.5.2.4.3 Records Section **??** introduced the AST representation of Records. As suggested by fig. 4.4, Nickel does not have AST nodes dedicated to record fields. Instead, it associates field names with values as part of the `Record` node. Since the language server is bound to process nodes individually, in effect, it will only see the values. Therefore, it can not process record values at the same time as the outer record. For the language server it is important to associate field names with their value, as it serves as name declaration. For that reason, NLS distinguishes `Record` and `RecordField` as independent kinds of linearization items where `RecordFields` act as a bridge between the record and the value named after the field.

To maintain similarity to other binding types, NLS has to create a separate item for the field and the value. This also ensures that the value can be linearized independently.

Record values may reference other fields defined in the same record regardless of the order, as records are recursive by default. Consequently, all fields have to be in scope and as such be linearized beforehand. When linearizing a record, NLS will generate `RecordField` items for each field. However, it can not associate the field's value with the item yet (which is expressed using `ValueState::None`). This is because the subtree of each field can be arbitrary large, as is the offset of the corresponding linearization items.

The visualization (fig. 4.5) of the record discussed in lst. 4.10 gives an example for this. Here, the first items linearized are record fields. Yet, as the `containers` field value is processed first, the `metadata` field value is offset by a number of fields unknown when the outer record node is processed.

To provide the necessary references, NLS makes use of the *scope safe* memory of its `Linearizer` implementation. This is possible, because each record value corresponds to its own scope. The complete process looks as follows:

1. When registering a record, first the outer `Record` is added to the linearization
2. This is followed by `RecordField` items for its fields, which at this point do not reference any value.
3. NLS then stores the `id` of the parent as well as the fields and the offsets of the corresponding items (`n-4` and `[(apiVersion, n-3), (containers, n-2), (metadata, n-1)]` respectively in the example fig. 4.5).
4. The `scope` method will be called in the same order as the record fields appear. Using this fact, the `scope` method moves the data stored for the next evaluated field into the freshly generated `Linearizer`
5. **(In the sub-scope)** The `Linearizer` associates the `RecordField` item with the (now known) `id` of the field's value. The cached field data is invalidated such that this process only happens once for each field.

4.5.2.4.4 Variable Reference The usage of a variable is always expressed as a `Var` node that holds an identifier. Registering a name usage is a multi-step process.

First, NLS tries to find the identifier in its scope-aware name registry. If the registry does not contain the identifier, NLS will linearize the node as `Unbound`. In the case that the registry lookup succeeds, NLS retrieves the referenced `Declaration` or `RecordField`.

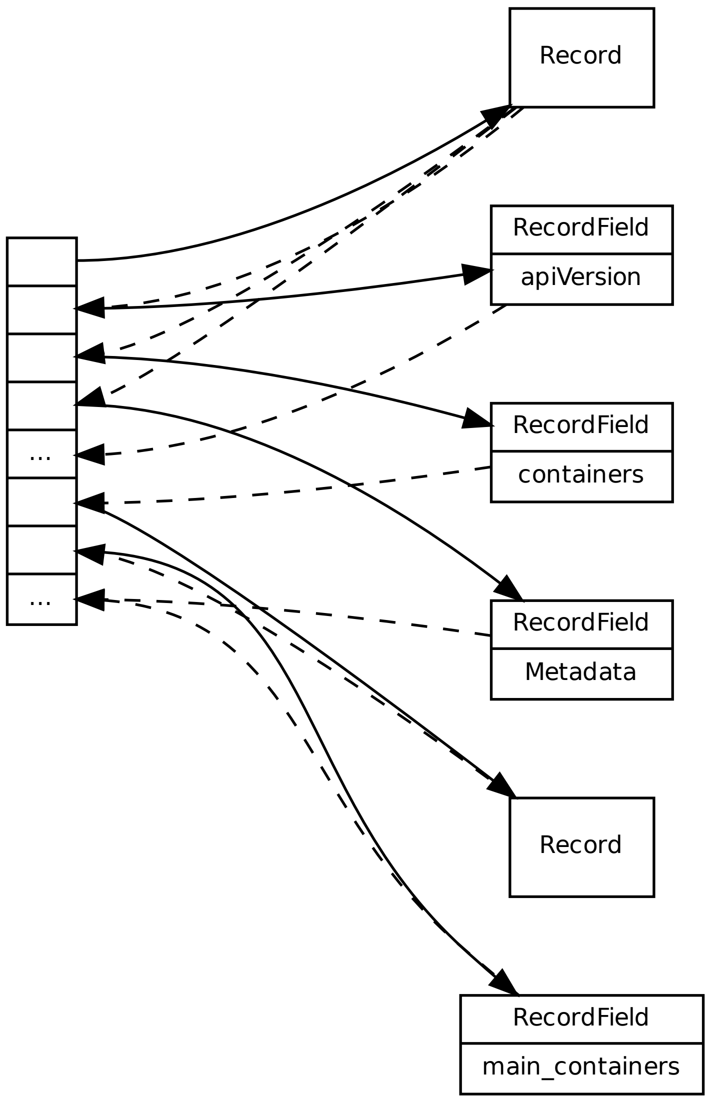


Figure 4.5: Linearization of a record

The linearizer will then add a usage item in the `Resolved` state to the linearization and update the declaration's list of usages.

4.5.2.4.5 Resolution of Record Fields

The AST representation of record destructuring in fig. 2.2 shows that accessing inner records involves chains of unary operations *ending* with a reference to a variable binding. Each operation encodes one field of a referenced record. However, to reference the corresponding declaration, the final usage has to be known. Therefore, instead of linearizing the intermediate elements directly, the `Linearizer` adds them to a shared stack until the grounding variable reference is registered.

Whenever a variable usage is linearized, NLS checks the stack for latent destructors (record accesses). If destructors are present, it adds `Usage` items for each element on the stack. Yet, because records are recursive it is possible that fields reference other fields' values.

Consider the following example lst. 4.11, which is depicted in fig. 4.6

Listing 4.11 Example of a recursive record

```
{
  y = {
    yy = "foo",
    yz = z,
  },
  z = y.yy
}
```

Here, a conflict is guaranteed. As the `Linearizer` processes the field values sequentially in arbitrary order, it is unable to resolve both `y.yz` and `z`.

Assuming the value for `z` is linearized first, the items corresponding the destructuring of `y` can not be resolved. While the *field* `y` is known, its value is not (cf. sec. 4.5.2.4.3), from which follows that `yy` is inaccessible. Yet, `y.yy` will be possible to resolve once the value of `y` is processed. For this reason the `Usage` generated from the destructor `.yy` is marked as `Deferred` and will be fully resolved during the post-processing phase as documented in sec. 4.5.3.2.

In fact, NLS linearized all destructor elements as `Deferred` and resolves the correct references later. Figure 4.7 shows this more clearly for the expression `x.y.z`. The `Declaration` for `x` is known, therefore its `Var` AST node is linearized as a `Resolved` usage. Mind that in records `x` could as well be a `RecordField`.

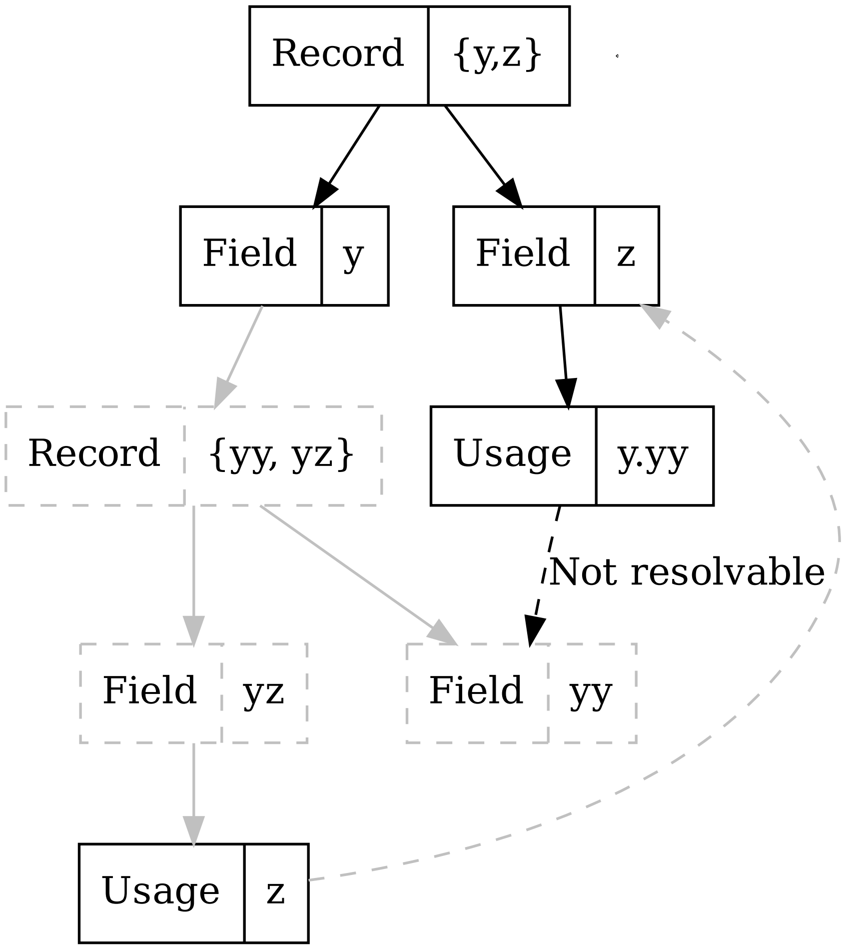


Figure 4.6: Example lock in recursive records. The field ‘y.yz‘ cannot be not be referenced at this point as the ‘y‘ branch has yet to be linearized

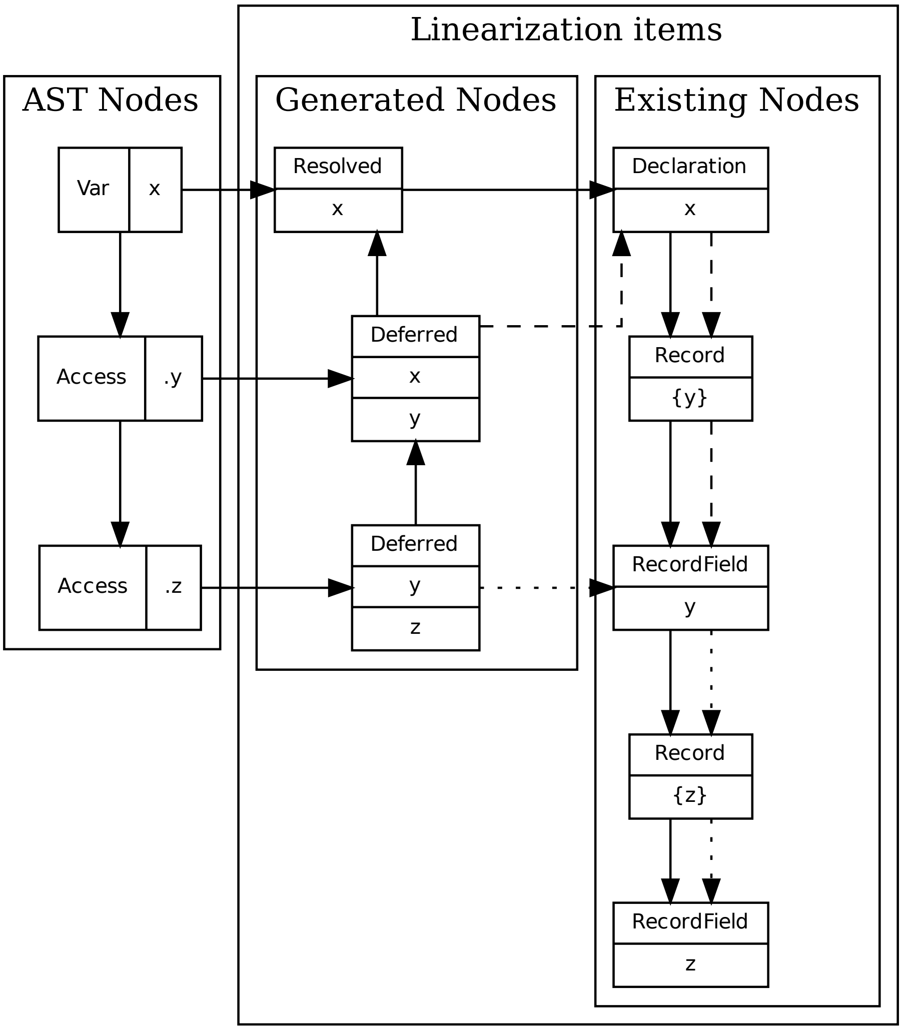


Figure 4.7: Depiction of generated usage nodes for record destructuring

4.5.2.4.6 Metadata In sec. 2.2.5.1.4 was shown that on the syntax level, metadata "wraps" the annotated value. Conversely, NLS encodes metadata as part of the `LinearizationItem` as it is considered to be intrinsically related to a value. NLS therefore has to defer handling of the `MetaValue` node until the processing of the associated value in the succeeding call. Like record destructors, NLS temporarily stores this metadata in the `Linearizer`'s memory.

Metadata always precedes its value immediately. Thus, whenever a node is linearized, NLS checks whether any latent metadata is stored. If there is, it moves it to the value's `LinearizationItem`, clearing the temporary storage.

Although metadata is not linearized as is, contracts encoded in the metadata can however refer to locally bound names. Considering that only the annotated value is type-checked and therefore passed to NLS, resolving Usages in contracts requires NLS to separately walk the contract expression. Therefore, NLS traverses the AST of expressions used as value annotations. In order to avoid interference with the main linearization, contracts are linearized using their own `Linearizer`.

4.5.3 Post-Processing

Once the entire AST has been processed NLS modifies the Linearization to make it suitable as an efficient index to serve various LSP commands.

After the post-processing the resulting linearization

1. allows efficient lookup of elements from file locations
2. maintains an `id` based lookup
3. links deeply nested record destructors to the correct definitions
4. provides all available type information utilizing Nickel's typing backend

4.5.3.1 Sorting

Since the linearization is performed in a preorder traversal, processing already happens in the order elements are defined in the file. Yet, during the linearization the location might be unstable or unknown for different items. Record fields for instance are processed in an arbitrary order rather than the order they are defined. Moreover, for nested records and record short notations, symbolic `Record` items are created which cannot be mapped to the original source and are thus placed at the range `[0..=0]` in the beginning of the file. Maintaining constant insertion performance and item-referencing requires that the linearization is exclusively appended. Given the examples above, this breaks the original order of the items with respect to their assigned position.

NLS thus defers reordering of items. The language server uses a stable sorting algorithm to sort items by their associated span's starting position. This way, nesting of items with the same start location is preserved. Since several operations require efficient access to elements by `id`, which after the sorting does not correspond to the items index in the linearization, after sorting NLS creates an index mapping `ids` to the new actual indices.

4.5.3.2 Resolving deferred access

Section **??** introduced the `Deferred` type for `Usages`. Resolution of usages is deferred if chained destructors are used. This is especially important in recursive records where any value may refer to other fields of the record which could still be unresolved.

As seen in fig. 4.7, the items generated for each destructor only link to their parent item. Yet, the root access is connected to a known declaration. Since at this point all records are fully processed NLS is able to resolve destructors iteratively.

First NLS collects all deferred usages in a queue. Each usage contains the *`id`* of the parent destructor as well as the *name* of the field itself represents. NLS then tries to resolve the base record for the usage by resolving the parent. If the value of the parent destructor is not yet known or a deferred usage, NLS will enqueue the destructor once again to be processed again later. In practical terms that is after the other fields of a common record. In any other case the parent consequently has to point to a record, either directly, through a record field or a variable. NLS will then get the `id` of the `RecordField` for the destructors *name* and mark the `Usage` as `Known` If no field with that name is present or the parent points to a `Structure` or `Unbound` usage, the destructor cannot be resolved in a meaningful way and will thus be marked `Unbound`.

4.5.3.3 Resolving types

Nickel features type inference in order to relieve the programmer of the burden of writing a lot of redundant type annotations. In a typed block, the typechecker is able to guess the type of all the values, even when they are not explicitly annotated by the user. To do so, the typechecker generates constraints derived from inspecting the AST, and solve them along the way. As a consequence, when a node is first encountered by NLS, its type is not necessarily known. There, the typechecker associate to the new node a so-called unification variable, which is a placeholder for a later resolved type. This unification variable is handed down to the `Linearizer`.

Similar to runtime processing, NLS needs to resolve the final types separately. After typechecking, NLS is eventually able to query the resolved type corresponding to an AST node using the associated unification variable.

4.5.4 Resolving Elements

4.5.4.1 Resolving by position

As part of the post-processing step discussed in sec. 4.5.3, the `LinearizationItems` in the completed linearization are reorderd by their occurence of the corresponding AST node in the source file. To find items in this list three preconditions have to hold:

1. Each element has a corresponding span in the source
2. Items of different files appear ordered by `FileId`
3. Two spans are either within the bounds of the other or disjoint.

$$\text{Item}^2_{\text{start}} \geq \text{Item}^1_{\text{start}} \land \text{Item}^2_{\text{end}} \leq \text{Item}^1_{\text{end}}$$

4. Items referring to the spans starting at the same position have to occur in the same order before and after the post-processing. Concretely, this ensures that the tree-induced hierarchy is maintained, more precise elements follow broader ones

This first two properties are an implication of the preceding processes. All elements are derived from AST nodes, which are parsed from files retaining their position. Nodes that are generated by the runtime before being passed to the language server are either ignored or annotated with synthetic positions that are known to be in the bounds of the file and meet the second requirement. For all other nodes the second requirement is automatically fulfilled by the grammar of the Nickel language. The last requirement is achieved by using a stable sort during the post-processing.

The algorithm used is listed in lst. 4.12. Given a concrete position, that is a `FileId` and `ByteIndex` in that file, a binary search is used to find the *last* element that *starts* at the given position. According to the aforementioned preconditions an element found there is equivalent to being the most specific element starting at this position. In the more frequent case that no element starting at the provided position is found, the search instead yields an index which can be used as a starting point to iterate the linearization *backwards* to find an item with the shortest span containing the queried position. Due to the third requirement, this reverse iteration can be aborted once an item's span ends before the query. If the search has to be aborted, the query does not have a corresponding `LinearizationItem`.

Listing 4.12 Resolution of item at given position

```
impl Completed {
  pub fn item_at(
    &self,
    locator: &(FileId, ByteIndex),
  ) -> Option<&LinearizationItem<Resolved>> {
    let (file_id, start) = locator;
    let linearization = &self.linearization;
    let item = match linearization
      .binary_search_by_key(
        locator,
        |item| (item.pos.src_id, item.pos.start))
      {
        // Found item(s) starting at `locator`
        // search for most precise element
        Ok(index) => linearization[index..]
          .iter()
          .take_while(|item| (item.pos.src_id, item.pos.start) == locator)
          .last(),
        // No perfect match found
        // iterate back finding the first wrapping linearization item
        Err(index) => {
          linearization[..index].iter().rfind(|item| {
            // Return the first (innermost) matching item
            file_id == &item.pos.src_id
            && start > &item.pos.start
            && start < &item.pos.end
          })
        }
    };
    item
  }
}
```

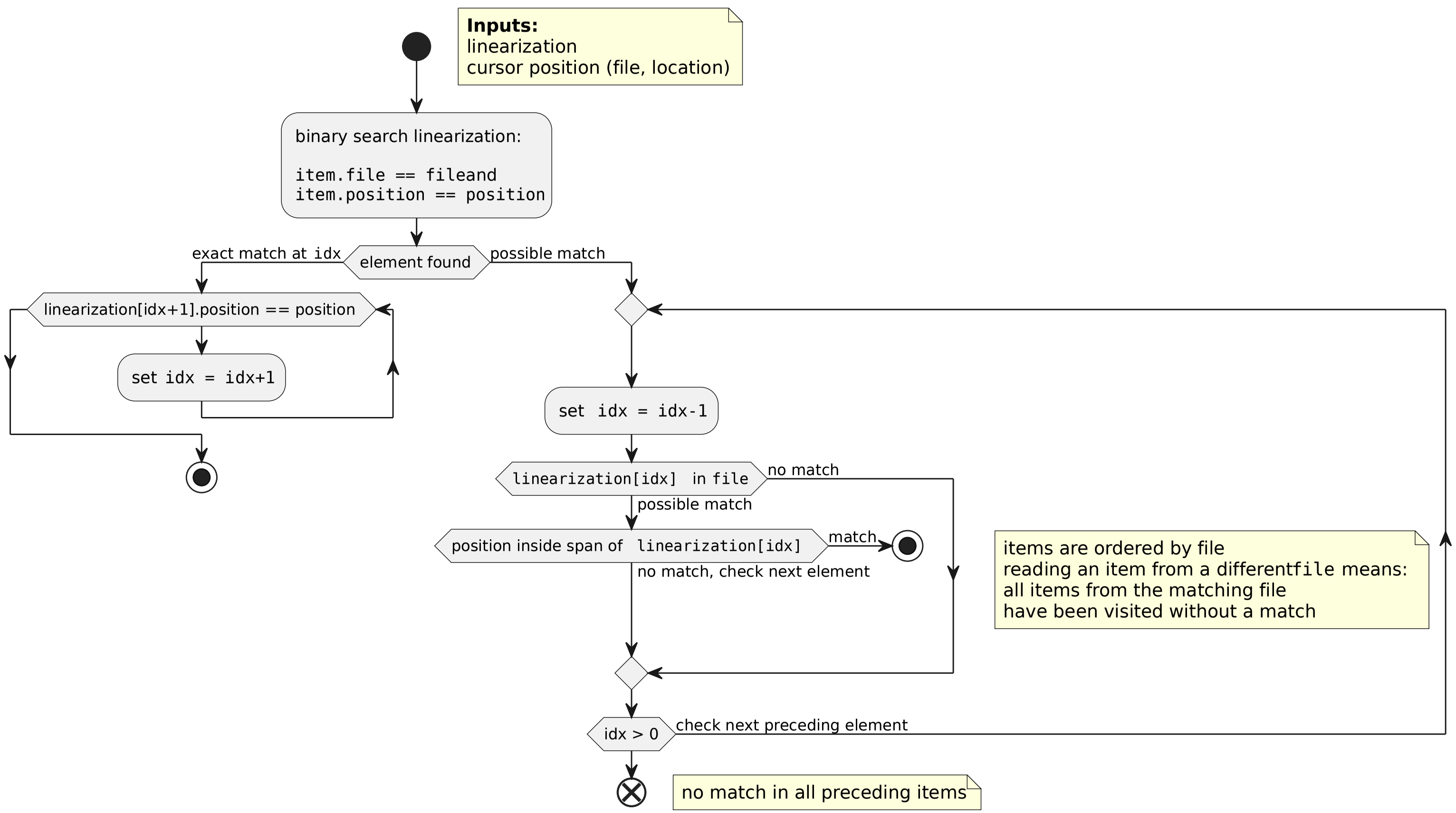


Figure 4.8: Activity diagram of item resolution by position

4.5.4.2 Resolving by ID

During the building process item IDs are equal to their index in the underlying array which allows for efficient access by ID. To allow similarly efficient dereferencing of node IDs, a `Completed` linearization maintains a mapping of IDs to their corresponding index in the reordered array.

4.5.4.3 Resolving by scope

During the construction from the AST, the syntactic scope of each element is eventually known. This allows to map an item's `ScopeId` to a list of elements defined in this scope by parent scopes. As discussed in sec. 4.5.2.2, scopes are lists of scope path elements, where the prefixes correspond to parent scopes. For instance, NLS would represent the example lst. 4.6 as shown in lst. 4.13 below.

Listing 4.13 Items collected for each scope of the example. Simplified representation using concrete values

```
/1 -> { Declaration("record") }
/1/1 -> { RecordField("key1"), RecordField("key2") }
/1/1/1 -> { "value" }
/1/1/2 -> { 123 }
/1/2 -> { Usage("record") }
```

For any given scope the set of referable nodes is determined by unifying the associated IDs of all prefixes of the given scope, then resolving the IDs to elements. Concretely, the identifiers in scope of the value `123` in the example 4.13 are `{Declaration("record"), RecordField("key1"), RecordField("key2") }`. The Rust implementation is given in lst. 4.14 below.

Listing 4.14 Resolution of all items in scope

```
impl Completed {
  pub fn get_in_scope(
    &self,
    LinearizationItem { scope, .. }: &LinearizationItem<Resolved>,
  ) -> Vec<&LinearizationItem<Resolved>> {
    let EMPTY = Vec::with_capacity(0);
    // all prefix lengths
    (0..scope.len())
      // concatenate all scopes
      .flat_map(|end| self.scope.get(&scope[..=end])
        .unwrap_or(&EMPTY))
      // resolve items
      .map(|id| self.get_item(*id))
      // ignore unresolved items
      .flatten()
      .collect()
  }
}
```

4.6 LSP Server Implementation

This section describes how NSL uses the linearization described in sec. 4.5 to implement the set of features proposed in sec. 4.1.2.

4.6.1 Server Interface

As mentioned in sec. 4.2.1 the Rust language ecosystem maintains several porjects supporting the development of LSP compliant servers. NLS is based on the `lsp-server` crate **lsp-server-crate?**, a contribution by the Rust Analyzer, which promises long-term support and compliance with the latest LSP specification.

Referring to fig. 4.3, the `Server` module represents the main server binary. It integrates the analysis steps with Nickel's parsing and type-checking routines. The resulting analysis is used to serve LSP requests.

4.6.2 Diagnostics and Caching

NLS instructs the LSP client to notify the server once the user opens or modifies a file. An update notification contains the complete source code of the file as well as its location. Upon notification, NLS first attempts to create an AST from the source code contained in the request payload by passing to Nickel's parser module. NLS will then instantiate a new linearizer which is applied to Nickel's type-checking functions, which has the following benefits. Leveraging type checking serves as both provider of type diagnostics or complete tree traversal yielding a linearization of the entire code in the absence of errors. Moreover, inferred types computed during the type-checking, can be used to resolve element types of the linearization items. Errors arising in either step reported to the client as Diagnostic including detailed information about location and possible details provided by the Nickel infrastructure.

As discussed in sec. 4.5 and sec. 4.5.4 the type-checking yields a `Completed` linearization which implements crucial methods to resolve elements. NLS will cache the linearization for each processed file so that it can provide its LSP functions even while a file is being edited, i.e, in a possibly invalid state.

4.6.3 Commands

Contrary to Diagnostics, which are part of a `Notification` based interaction with the client and thus entirely asynchronous, `Commands` are issued by the client which expects an explicit synchronous answer. While servers may report long-running tasks and defer sending eventual results back, user experience urges quick responses. NLS achieves the required low latency by leveraging the eagerly built linearization. Consequently, the language server implements most `Commands` through a series of searches and lookups of items.

4.6.3.1 Hover

When hovering an item or issuing the corresponding command in text based editors, the LSP client will send a request for element information containing the cursor's *location* in a given *file*. Upon request, NLS loads the cached linearization and performs a lookup for a `LinearizationItem` associated with the location using the linearization interface presented in sec. 4.5.4.1. If the linearization contains an appropriate item, NLS serializes the item's type and possible metadata into a response object which is resolves the RPC call. Otherwise, NLS signals no item could be found.

4.6.3.2 Jump to Definition and Show references

Similar to *hover* requests, usage graph related commands associate a location in the source with an action. NLS first attempts to resolve an item for the requested position using the cached linearization. Depending on the command the item must be either a `Usage` or `Declaration/RecordField`. Given the item is of the correct kind, the language server looks up the referenced declaration or associated usages respectively. The stored position of each item is encoded in the LSP defined format and sent to the client. In short, usage graph queries perform two lookups to the linearization. One for the requested element and a second one to retrieve the linked item.

4.6.3.3 Completion

Item completion makes use of the scope identifiers attached to each item. Since Nickel implements lexical scopes, all declarations made in parent scopes can be a reference. If two declarations use the same identifier, Nickel applies variable shadowing to refer to the most recent declaration, i.e., the declaration with the deepest applicable scope. NLS uses scope identifiers which represent scope depth as described in sec. 4.5.2.2 to retrieve symbol names for a reference scope using the method described in sec. 4.5.4.3. The current scope taken as reference is derived from the item at cursor position.

4.6.3.4 Document Symbols

The Nickel Language Server interprets all items of kind `Declaration` as document symbol. Accordingly, it filters the linearization by kind and serializes all declarations into an LSP response object.

5 Evaluation

Section ?? described the implementation of the Nickel Language Server addressing the first research question stated in sec. 1.2. Proving the viability of the result and answering the second research question demands an evaluation of different factors.

Earlier, the most important metrics of interest were identified as:

Usability What is the real-world value of the language server?
Does it improve the experience of developers using Nickel? NLS offers several features, that are intended to help developers using the language. The evaluation should assess whether the server does improve the experience of developers.
Does NLS meet its users' expectations in terms of completeness, correctness and behavior? Labeling NLS as a Language Server, invokes certain expectations built up by previous experience with other languages and language servers. Here, the evaluation should show whether NLS lives up to the expectations of its users.

Performance What are the typical latencies of standard tasks? In this context latency refers to the time it takes from issuing an LSP command to the reply of the server. The JSON-RPC protocol used by the LSP is synchronous, i.e. requires the server to return results of commands in the order it received them. Since most commands are sent implicitly, a quick processing is imperative to avoid commands queuing up.
Can single performance bottlenecks be identified? Single commands with excessive runtimes can slow down the entire communication resulting in bad user experience. Identified issues can guide the future work on the server.
How does the performance of NLS scale for bigger projects? With increasing project sizes the work required to process files increases as well. The evaluation should allow estimates of the sustained performance in real-world scenarios.

Answering the questions above, this chapter consists of two main sections. The first section sec. 5.2 introduces methods employed for the evaluation. In particular, it details the survey (sec. 5.3.1) which was conducted with the intent to gain qualitative opinions by users, as well as the tracing mechanism (sec. 5.3.2) for factual quantitative insights. Section 5.3 summarizes the results of these methods.

5.1 Evaluation Considerations

Different methods to evaluate the abovementioned metrics were considered. While quantifying user experience yields statistically sound insights about the studied subject, it fails to point out specific user needs. Therefore, this work employs a more subjective evaluation based on a standardized experience report focusing on individual features. Contrasting the expectations highlights well executed, immature or missing features. This allows more actionable planning of the future development to meet user expectations.

On the other hand it is more approachable to track runtime performance objectively through time measurements. In fact, runtime behavior was a central assumption underlying the server architecture. As discussed in sec. **??** NLS follows an eager, non-incremental processing model. While incremental implementations are more efficient, as they do not require entire files to be updated, they require explicit language support, i.e., an incremental parser and analysis. Implementing these functions exceeds the scope of this work. Choosing a non-incremental model on the other hand allowed reusing entire modules of the Nickel language. The analysis itself can be implemented both in a lazy or eager fashion. Lazy analysis implies that the majority of information is resolved only upon request instead of ahead of time. That is, an LSP request is delayed by the analysis before a response is made. Some lazy models also support memorizing requests, avoiding recomputing previously requested values. However, eager approaches preprocess the file ahead of time and store the analysis results such that requests can be handled mostly through value lookups. To fit Nickels' type-checking model and considering that in a typical Nickel workflow, the analysis should still be reasonably efficient, the eager processing model was chosen over a lazy one.

5.2 Methods

5.2.1 Objectives

The qualitative evaluation was conducted with a strong focus on the first metric in [sec:metrics]. Usability proves hard to quantify, as it is tightly connected to subjective perception, expectations and tolerances. The structure of the survey is guided by two additional objectives, endorsing the separation of individual features. On one hand, the survey should inform the future development of NLS; which feature has to be improved, which bugs exist, what do users expect. This data is important for NLS both as an LSP implementation for Nickel (affecting the perceived maturity of Nickel) and a generic basis for other projects. On the other hand, since all features are essentially queries to the common linearization data structure (cf. sec. **??**), the implementation of this central structure is an essential consideration. The survey should therefore also uncover apparent problems with this architecture. This entails the use of language abstractions (cf. sec. 4.5) and the integration of Nickel core functions such as the type checking procedure.

The quantitative study in contrast focuses on measurable performance. Similarly to the survey-based evaluation, the quantitative study should reveal insight for different features and tasks separately. The focus lies on uncovering potential spikes in latencies, and making empirical observations about the influence of Nickel file sizes.

5.2.2 Qualitative Evaluation Setup {#sec:qualitative@methods}

Inspired by the work of Leimeister in **leimeister?**, a survey aims to provide practical insights into the experience of future users. In order to get a clear picture of the users' needs and expectations independently of the experience, the survey consists of two parts – a pre-evaluation and final survey.

5.2.2.1 Pre-Evaluation

5.2.2.1.1 Expected features The pre-evaluation introduced participants in brief to the concept of language servers and asked them to write down their understanding of several LSP features. In total, six features were surveyed corresponding to the implementation as outlined in sec. 4.1.2, namely:

5.2.2.1.2 Expected behavior

1. Code completion Suggest identifiers, methods or values at the cursor position.
2. Hover information Present additional information about an item under the cursor, i.e., types, contracts and documentation.
3. Jump to definition Find and jump to the definition of a local variable or identifier.
4. Find references List all usages of a defined variable.
5. Workspace symbols List all variables in a workspace or document.
6. Diagnostics Analyze source code, i.e., parse and type check and notify the LSP Client if errors arise. The item for the "Hover" feature for instance reads as follows:

    > Editors can show some additional information about code under the cursor. The selection, kind, and formatting of that information is left to the Language Server.
    >
    > What kind of information do you expect to see when hovering code? Does the position or kind of element matter? If so, how?

Items first introduce a feature on a high level followed by asking the participant to describe their ideal implementation of the feature.

5.2.2.2 Experience Survey

For the final survey, interested participants at Tweag were invited to a workshop introducing Nickel. The workshop allowed participants unfamiliar with the Nickel language to use the language and experience NLS in a more natural setting. Following the workshop, participants filled in a second survey which focused on three main aspects:

First, the general experience of every individual feature. Without weighing their in expectations, the participants were asked to give a short statement of their experience.

The item consists of a loose list of statements with the aim to achieve a rough quality classification:

☐ The feature did not work at all
☐ The feature behaved unexpectedly
☐ The feature did not work in all cases
☐ The feature worked without an issue
☐ Other

The following items survey the perceived performance and stability. The items were implemented as linear scales that span from "Very slow response" to "Very quick response" and "Never Crashed" to "Always Crashed" respectively. The second category asked participants to explicitly reflect on their expectations:

○ The feature did not work at all
○ Little of my expectation was met
○ Some expectations were met, enough to keep using NLS for this feature
○ Most to all expectations were met ○ NLS surpassed the expectations ○ Other

In the final part participants could elaborate on their answers.

Why were they (not) satisfied?
What is missing, what did they not expect?

5.2.3 Quantitative {#sec:quantitative@methods}

To address the performance metrics introduced in sec. **??**, a quantitative study was conducted, that analyzes latencies in the LSP-Server-Client communication. The study complements the subjective reports collected through the survey (cf. sec. 5.2.2.2). The evaluation is possible due to the inclusion of a custom tracing module in NLS. The tracing module is used to create a report for every request, containing the processing time and a measure of the size of the analyzed document. If enabled, NLS records an incoming request with an identifier and time stamp. While processing the request, it adds additional data to the record, i.e., the type of request, the size of the linearization (cf. sec. 4.5) or processed file and possible errors that occurred during the process. Once the server replies to a request, it records the total response time and writes the entire record to an external file.

The tracing approach narrows the focus of the performance evaluation to the time spent by NLS. Consequently, the performance evaluation is independent of the LSP client (editor) that is used. Unlike differences in hardware which affects all operations similarly, LSP clients may implement different behaviors that may cause editor-specific biases. For instance, the LSP does not specify the frequency at which file changes are detected, which in turn can lead to request queuing depending on the editor used.

5.3 Results

5.3.1 Qualitative

As outlined in [#sec:qualitative-study-outline], the qualitative study consists of two parts conducted before and after an introductory workshop. The pre-evaluation aimed to catch the users' expected features and behaviors, while the main survey asked users about their concrete experiences with the NLS.

5.3.1.1 Pre-Evaluation

In the initial free assessment of expected features (c.f. [#sec:expected-features]) the participants unanimously identified four of the six language server capabilities that guided the implementation of the project (c.f. sec. 2.1.2): Type-information on hover, automatic diagnostics, Code Completion and Jump-to-Definition.

The other two features, Find-References and Workspace/Document Symbols on the contrary were sparingly commented. Some participants noted that they did not use these capabilities.

5.3.1.1.1 Type-information on hover Hovering is expected to work on values as well as functions. For values, it is desired to show types including applied contracts, documentation and default values. On functions, it should display the function's signature and documentation. Additionally, hovering an item desirably visualizes the scope of the item, i.e. where it is available.

5.3.1.1.2 Diagnostics Diagnostics are expected to include error messages signalling syntax and type errors as well as possibly evaluation errors and contract breaches. The diagnostics should show up at the correct positions in the code and "suggest how to fix" mistakes. Code linting was named as a possible extension to error reporting. This would include warnings about bad code style – formatting, casing conventions – unused variables, deprecated code and undocumented elements. Moreover, structural analysis was conceived to allow finding structural issues and help to fix them In either case the diagnostic should be produced "On-the-fly" while typing or upon saving the document.

5.3.1.1.3 Code Completion Code Completion was described as a way to chose from possible completion candidates of options. Completable items can be variable names, record fields, types or functions. Besides, Participants conceived filtering or prioritizing of candidates by type if applied as function arguments. Finally, the completion context could guide prioritization as well as auto-generation of contract and function skeletons.

5.3.1.1.4 Jump-to-Definition Users expect Jump-to-Definition to work with any kind of reference i.e., variable usages, function calls, function arguments and type annotations. On records and references to records, users expect statically defined nested fields to point to the correct respective definition. The ability to define self referencing records was however conceded to be a challenge.

5.3.1.1.5 Other features Syntax highlighting and code formatting as well as error tolerance were named as further desirable features of a language server beyond the explicitly targeted features. Error tolerance was detailed as the capability of the language server to continue processing and delivering analysis of invalid sources. For invalid files a language server should still be able to provide its functionality for the correct parts of the program.

5.3.1.2 Experience Survey

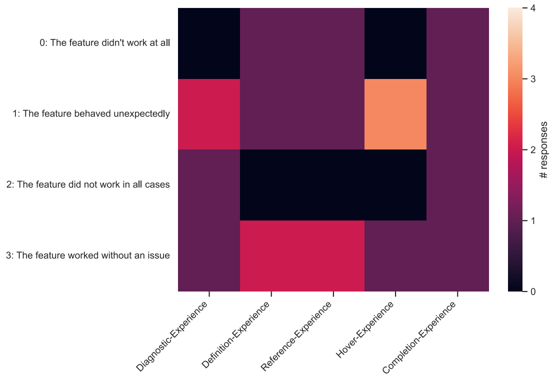
This subsection describes the results from the filled after the Nickel workshop in which participants were asked to install the LSP to support their experience. It first looks at a summary of the data, before diving into the comments for each directly addressed feature.

The above figures show the turnout of three items from the survey for each of the relevant features. Neither of them shows clear trends with positive and negative results distributed almost evenly between positive and negative sentiment.

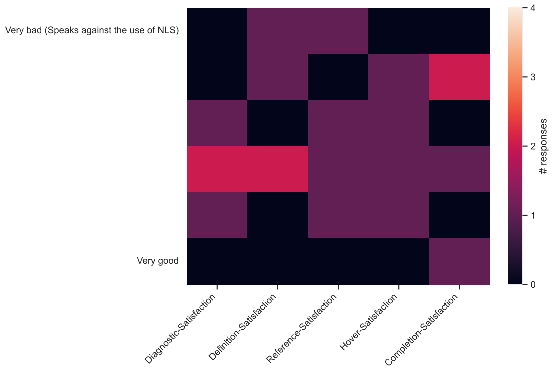
The first graph (fig. 5.1a) represents the participants' general experience with the relevant features. It shows that each feature worked without issue in at least one instance. Yet, three features were reported to not work at all and no feature left the users unsurprised. Users found the hover and diagnostic features to behave particularly unexpectedly.

In the second item of each feature, the survey asked the subjects to rate the quality of the language server based on their expectations. Figure 5.1c summarizes the results. In agreement with the first graph, one user was unable to use at least three features entirely. The majority of responses show that NLS met its user's expectations at least partially. The results are however highly polarized as the Jump-to-Definition and Hover features demonstrate; Each received equally many votes for being inapt and fully able to hold up to the participants expectations at the same time. Other features were left with a uniformly distributed assessment (e.g. Completion and Find-References). The clearest result was achieved by the Diagnostics feature, which received a slight but uncontested positive sentiment.

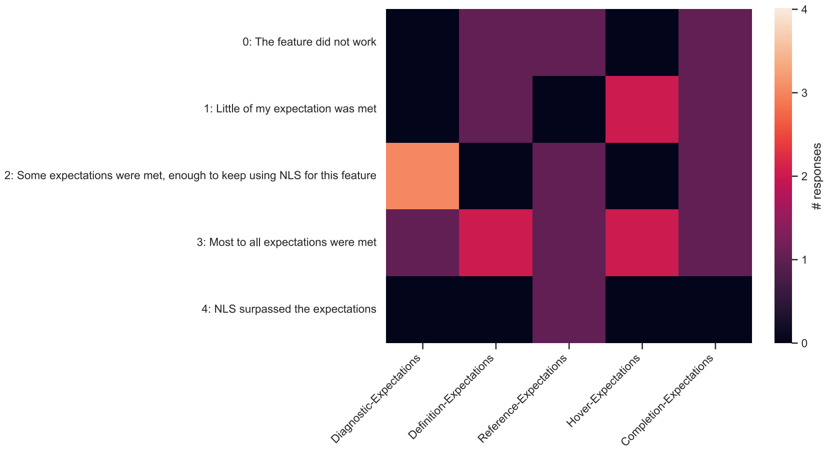
The general satisfaction with each feature was answered in the same polarized manner as seen in fig. 5.1b. A slight majority of responses falls into the upper half of the possible spectrum. Two of the features reported without function in the preceding questions were given the lowest possible rating.



(a) Number of votes for each experience category and feature



(b) Number of votes for each satisfaction category and feature



(c) Number of votes for each expectation category and feature

Figure 5.1: User responses regarding general experience, fulfillment of expectations and general satisfaction.

5.3.1.2.1 Hover {#sec:hover@res} As apparent in (fig. 5.1a), most participants experienced unexpected behavior by the LSP when using the hover functionality. In the comments, extraneous debug output and incorrect displaying of the output by the IDE are pointed out as concrete examples. However, one answer suggests that the feature was working with "usually useful" output.

5.3.1.2.2 Diagnostics {#sec:diagnostics@res} While the diagnostics shown by NLS appear to behave unexpectedly for some users in fig. 5.1a, no user felt deterred from keep using NLS for it as displayed in fig. 5.1c. Some respondents praised the "quick" and "direct feedback" as well as the visual error markers pointing to the exact locations of possible issues. On the contrary, others mentioned "unclear messages" and pointed out that contracts were not checked by the Language Server. Moreover, a performance issue was brought up noting that in some situations NLS "queues a lot of work and does not respond."

5.3.1.2.3 Code Completion {#sec:code-completion@res} Comments about the Code Completion feature were unanimously critical. Some participants noted the little gained "value over the token based completion built into the editor" while others specifically pointed at "missing type information and docs." Additionally, record field completion was found to be missing, albeit highly valued.

5.3.1.2.4 Document Navigation {#sec:document-navigation@res} Results and comments about the Go-To-Definition and Find-References were polarized. Some users experienced unexpected behavior or were unable to use the feature at all (cf. fig. 5.1a). Similarly, the comments on one hand suggest that "the feature works well and is quick" while on the other mention inconsistencies and unavailability. More specifically, cross file navigation was named an important missing feature.

5.3.1.2.5 General Performance {#sec:general-performance@res} The responses to the general performance suggest that NLS' performance is largely dependent on its usage. On unmodified files queries were reported to evaluate "instantaneously." However, modifying files caused that "modifications stack up" causing high CPU usage and generally "very slow" responses. Besides, documentation was reported as slow to resolve while the server itself was "generally fast."

5.3.2 Quantitative

The quantitative evaluation focuses on the performance characteristics of NLS. As described in sec. **??** a tracing module was embedded into the NLS binary which recorded the runtime together with the size of the analyzed data, i.e., the number of linearization items sec. 4.5 or size of the analyzed file. This section will first introduce the dataset before looking at the general performance and finally looking into particular cases.

5.3.2.1 Dataset

The underlying data set consists of 16760 unique trace records. Since the `textDocument/didOpen` method is executed on every update of the source, it greatly outnumbers the other events. The final distribution of methods traced is:

Table 5.1: Number of traces per LSP method

Method	count	linearization based
`textDocument/didOpen`	13436	no
`textDocument/completion`	2981	yes
`textDocument/hover`	227	yes
`textDocument/definition`	68	yes
`textDocument/references`	49	yes
total	16761	

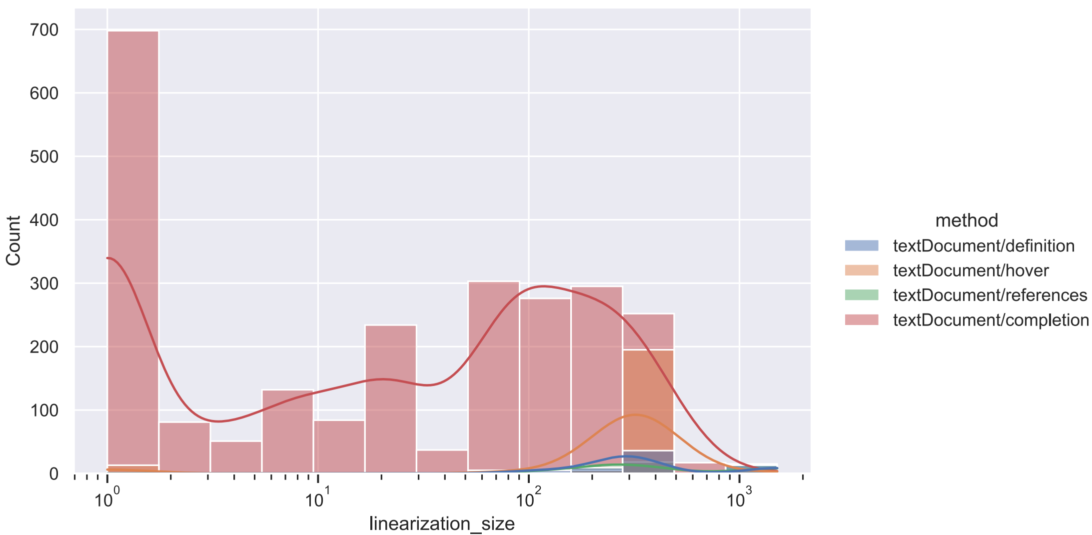


Figure 5.2: Distribution of linearization based LSP requests

Figures 5.2 break up these numbers by method and linearization size or file size respectively. The linearization is the linear representation of an enriched AST. It is explained in great detail in sec. 4.5. The first figure shows a peak number of traces for completion events between 0 to 1 linearization items as well as local maxima around a linearization size of

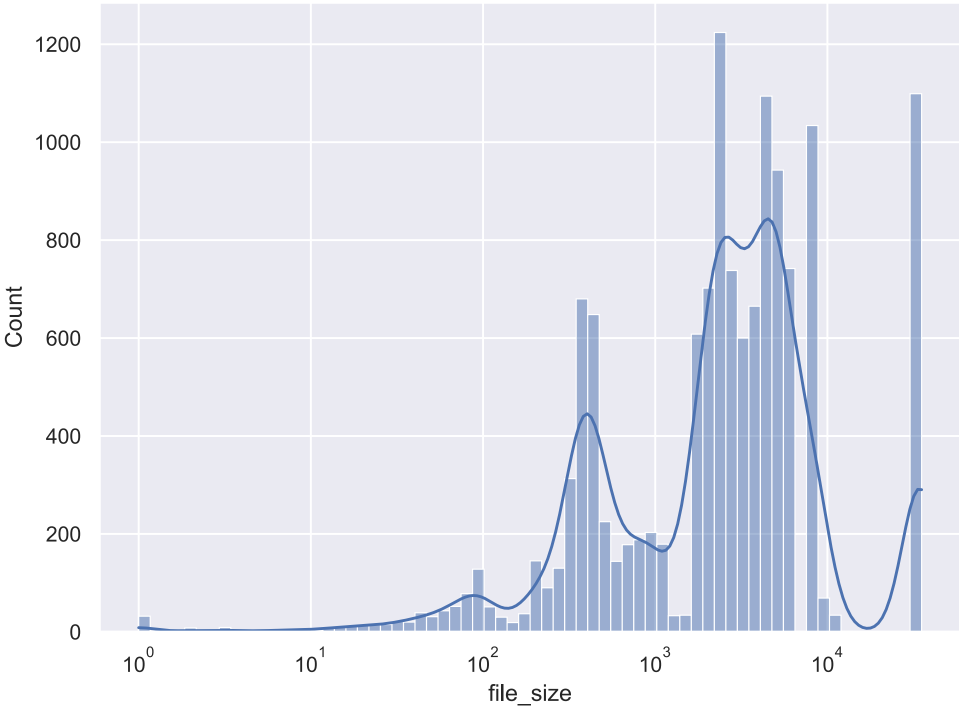


Figure 5.3: Distribution of file analysis requests

20 to 30 and sustained usage of completion requests in files of $90 - 400$ items. Similar to the completion requests (but well outnumbered in total counts), other methods were used mainly in the range between 200 and 400 linearization items. A visualization of the Empirical Cumulative Distribution Function (ECFD) fig. **??** corroborates these findings. Moreover, it shows an additional hike of Jump-to-Definition and Find-References calls at on files with around 1500 linearization items. The findings for linearization based methods line up with those depicting linearization events (identified as `textDocument/didOpen`). An initial peak referring to rather small input files between 300 and 400 bytes in size is followed by a sustained usage of the NLS on files with 2 to 6 kilobytes of content topped with a final application on 35 kilobyte large data.

5.3.2.2 Big Picture Latencies

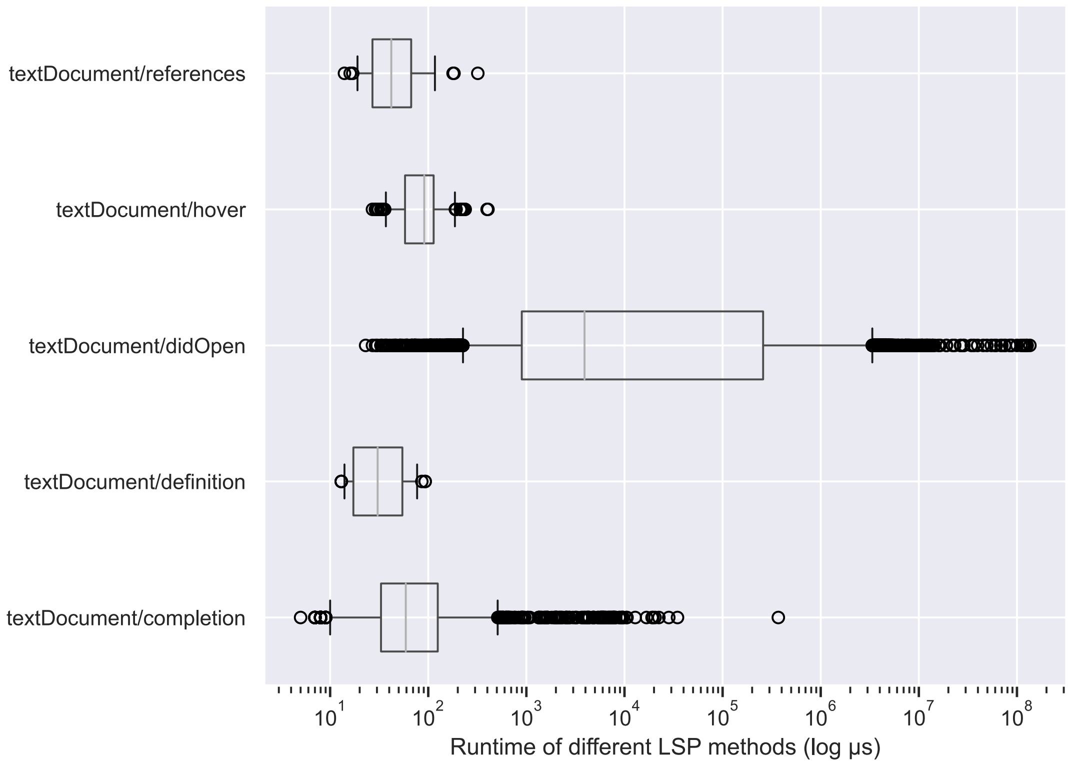


Figure 5.4: Statistical runtime of different LSP methods

Comparing the runtime of the individual methods alone in fig. 5.4, reveals three key findings. First, all linearization based methods exhibit a sub-millisecond latency in at least 95 of all invocations and median response times of less than 100ţs However, maximum latencies of completion invocations reached tens of milliseconds and in one recorded case about $300ms$. Finally, document linearization as associated with the `textDocument/didOpen` method shows a great range with maxima of $1.5 * 10^5$ţs (about 2.5 minutes) and a generally greater interquartile range spanning more than two orders of magnitude.

5.3.2.3 Special cases

Setting the runtime of completion requests in relation to the linearization size on which the command was performed shows no clear correlation between the dimensions. In fact the correlation coefficient between both variables measures 0.01617 on a linear scale and 0.26 on a $\log_{10}\log_{10}$ scale. Instead, vertical columns stand out in the correlation graph fig. 5.5a. The height of these columns varies from one to five orders of magnitude. The item density shows that especially high columns form whenever the server receives a higher load of requests. Additionally, color coding the individual requests by time reveals that the trace points of each column were recorded at a short time interval. Applying the same analysis to the other methods in figs. 5.5b, 5.5c, **??** returns similar findings, although the columns remain more compact in comparison to the Completions method. In case of the `didOpen` method columns are clearly visible too [#fig:correlation-opens]. However, here they appear leaning as suggesting an increase in computation time as the file grows during a single series of changes to the file.

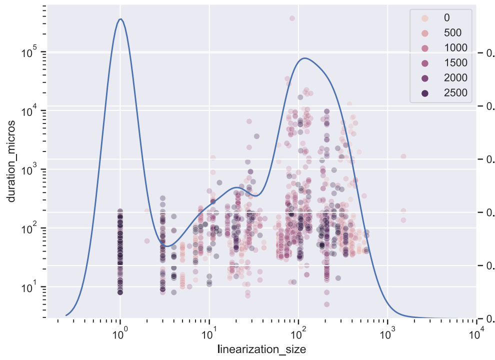
5.4 Discussion

This section discusses the issues raised during the survey and uncovered through the performance tracing. In the first part the individual findings are summarized and if possible grouped by their common cause. The second part addresses each cause and connects it to the relevant architecture decisions, while explaining the reason for it and discussing possible alternatives.

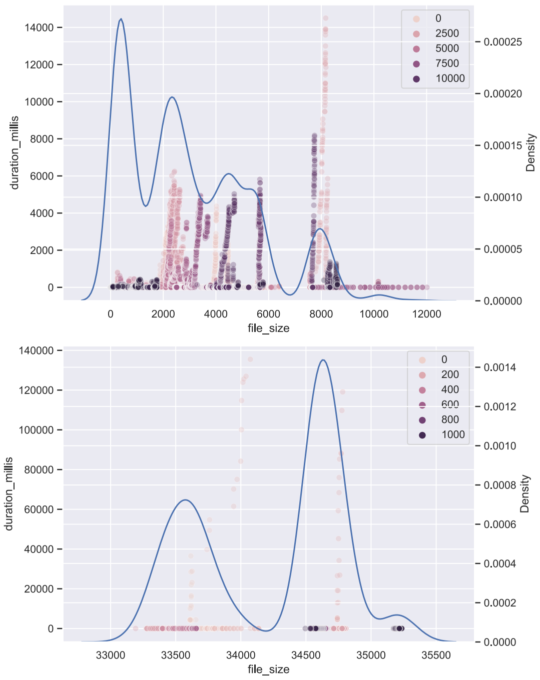
During the qualitative evaluation several features did not meet the expectations of the users. The survey also hinted performance issues that were solidified by the results of the quantitative analysis.

5.4.1 Diagnostics

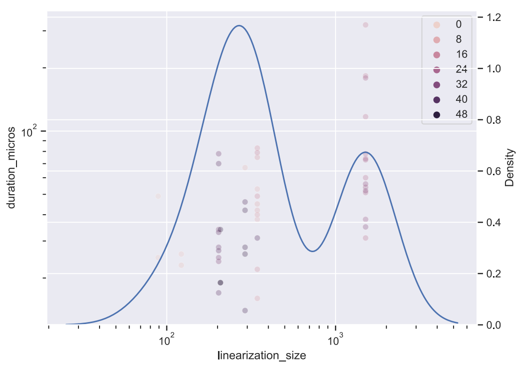
First, participants criticized sec. 2.1.3.6 the diagnostics feature for some unhelpful error messages and specifically for not taking into account Nickel's hallmark feature, Contracts sec. **??**. While Contracts are a central element of Nickel and relied upon to validate data, the language server does not actually warn about contract breaches. Yet, while contracts and their application looks similar to types, contracts are a dynamic language element which are dynamically applied during evaluation. Therefore, it is not possible to



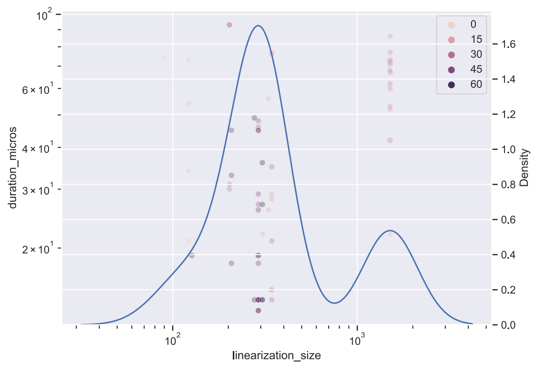
(a) Runtime latencies of completion requests at different linearization sizes



(b) Runtime latencies of hover requests at different linearization sizes



(c) Runtime latencies of find-references requests at different linearization sizes



(d) Runtime latencies of jump-to-definition requests at different linearization sizes

Figure 5.5: Runtime latencies of different linearization based methods

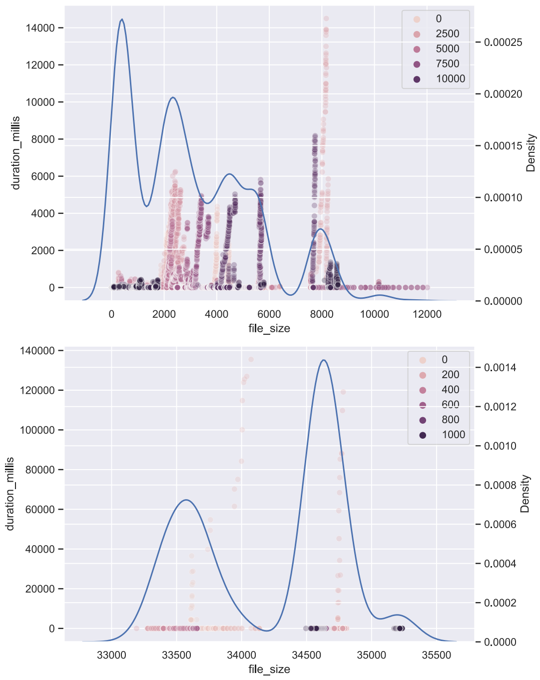


Figure 5.6: Runtime latencies of file update handling at different file sizes

determine whether a value conforms to a contract without evaluation of the contract. NLS's is integrated with Nickel's type-checking mechanism which precedes evaluation and provides only a static representation of the source code. In order to support diagnostics for contracts NLS would need to locally evaluate arbitrary code that makes up contracts. However, contracts can not be evaluated entirely locally as they may transitively depend on other contracts. This is particularly true for a file's output value. Additionally, Contracts can implement any sort of complex computation including unbound recursion. Due to these caveats, evaluating contracts as part of NLS's analysis implies the evaluation of the entire code which was considered a possibly significant impact to the performance. As laid out above evaluating contracts locally is no option either. It is not only challenging to collect the minimal context of the Contract, the context may in fact be the entire program. An alternative option is to provide the ability to apply contracts manually using an LSP feature called "Code Lenses." Code Lenses are displayed by editors as annotations allowing the user to manually execute an associated action.

5.4.2 Cross File Navigation

In both cases `Jump-To-Definition` and `Find-References` surveyed users requested support for cross file navigation. In particular, finding the definition of a record field of an imported record should navigate the editor to the respective file as symbolized in lst. 5.1.

Listing 5.1 Minimal example of cross file referencing

```
// file_a.ncl

let b = import "./b.ncl" in b.field
                              |
                              +------+
                                     |
-----------------------------------  |
                                     |
// file_b.ncl                        |
                                     |
{                                    |
  field = "field value";             |
}  ^                                 |
   +---------------------------------+
```

The resolution of imported values is done at evaluation time, the AST therefore only contains nodes representing the concept of an import but no not reference elements of that file. NLS does ingest the AST without resolving these imports manually. The type checking module underlying NLS still recurses into imported files to check their formal correctness. As a result it would be possible for a NLS to resolve these links as an additional step in the post-processing by either inserting artificial linearization items sec. 4.5 or merging both files' linearization entirely.

5.4.3 Autocompletion

Another criticized element of NLS was the autocompletion feature. In the survey, participants mentioned the lack of additional information and distinction of elements as well as NLS inability to provide completion for record fields. In Nickel, record access is declared by a period. An LSP client can to configured to ask for completions when such an access character is entered additionally to manual requests by the user. The language server is then responsible to provide a list of completion candidates depending on the context, i.e. the position. [Section #sec:completion] describes how NLS resolves this kind of request. NLS just lists all identifiers of declarations that are in scope at the given position. Notably, it does not take the preceding element into account as additional context. To support completing records, the server must first be aware of separating tokens such as the period symbol, check whether the current position is part of a token that is preceded by a separator and finally resolve the parent element to a record.

5.4.4 Performance

In the experience survey performance was pointed out as a potential issue. Especially in connection with the diagnostics and hover feature. NLS was described to "queue a lot of work and not respond" and show different performance signatures depending on its usage. While commands resolved "instantaneously" on unmodified files, editing a file causes high CPU usage and generally "very slow" responses. An analysis of the measured runtime of 16761 requests confirmed that observation. Both Hover and Update requests showed a wide range of latencies with some reaching more than two minutes. However, the data distribution also confirmed that latencies for most requests except `didOpen` are distributed well below one millisecond. The `didOpen` requests which are associated with the linearization process sec. 4.5 peak around $1ms$ but longer latencies remain frequent fig. ??. Looking deeper into the individual features, reveals signs of the aforementioned "stacking." As discussed in sec. 5.3.2.3 subsequent requests exhibit increasing processing times especially during peak usage.

This behavior is caused by the architecture of the LSP and NLS' processing method. The Language Server Protocol is a synchronous protocol which requires the processing of all requests FIFO order. In effect, every request is delayed until previous requests are handled. This effect is particularly strong as the server is faced with a high volume. In the case of the trace for `didOpen` events the delay effect is greater than for other methods as `didOpen` is associated with a full analysis of the entire file. NLS architecture is heavily influenced by the desire to reuse as many elements of the Nickel runtime as possible to maintain feature parity with the evolving language core. Consequently, file updates invoke a complete eager analysis of the contents; The entire document is parsed, type checked and recorded to a linearization every time. In contrast, all other methods rely on the linearization of a document which allows them to use a binary search to efficiently lookup elements in logarithmic time. Additionally, all requests regardless of their type are subject to the same queue. Given that `didOpen` requests make up > 80 of the recorded events, suggests that other events are heavily slowed down collaterally.

Multiple ways exist to address this issue by reducing the average queue size. The most approachable way to reduce queue sizes is to reduce the number of requests the server needs to handle. The `didOpen` trace elements actually represents the joint processing path of initial file openings, and changes. NLS configures clients to signal changes both on save and following editor defined "change." The fact that it is the editor's responsibility to define what constitutes a change means that some editors send invoke the server on every key press. In fig. 5.6 signs for such a behavior can be seen as local increases of processing time as the document grows. Hence, restricting analysis to happen only as the user saves the document could potentially reduce the load of requests substantially. Yet, many users preferred automatic processing to happen while they type. To server this pattern, NLS could implement a debouncing mechanism for the processing of document changes. The messages associated to document changes and openings are technically no requests but notifications. The specification of JSON-RPC which the LSP is based on defines that notifications are not allowing a server response. Clients can not rely on the execution of associated procedures. In effect, a language server like NLS, where each change notification contains the entire latest document, may skip the processing of changes. In practice, NLS could skip such queue items if a more recent version of the file is notified later in the queue. The queue size can also be influenced by reducing the processing time. Other language servers such as the rust-analyzer **rust-analyzer?** chose to process documents lazily. Update requests incrementally change an internal model which other requests use as a basis to invoke targeted analysis, resolve elements and more. The entire model is based on in incremental computation model which automates memorization of requests. This method however requires rust-analyzer to reimplement core components of rust to support incrementally. Therefore, if one accepts to implement an incremental model of Nickel (including parsing and type checking) to enable incremental analysis in NLS, switching to a lazy model is a viable method to reduce the processing time of change notifications and shorten the queue.

6 Discussion

6.1 Project results

6.2 Project shortcomings

Future Work

[1] "JSON-RPC 2.0 Specification," JSON-RPC Working Group, Specification, Jan. 2013 [Online]. Available: https://www.jsonrpc.org/specification. [Accessed: Dec. 15, 2021]

[2] A. D. Birrell and B. J. Nelson, "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems*, vol. 2, no. 1, p. 21, 1984.

[3] Microsoft, "Language Server Protocol Specification - 3.16." Dec. 2020 [Online]. Available: https://microsoft.github.io/language-server-protocol/specification.html. [Accessed: Jul. 28, 2021]

[4] Sourcegraph, "Langserver.org." [Online]. Available: https://langserver.org/. [Accessed: Apr. 27, 2022]

[5] W3C, "Extensible Markup Language (XML) 1.0 (Fifth Edition)," Specification, Nov. 2008 [Online]. Available: https://www.w3.org/TR/2008/REC-xml-20081126/. [Accessed: May 24, 2022]

[6] P. Charollais, "ECMA-404, 2nd edition, December 2017," p. 16, 2017.

[7] "YAML Ain't Markup Language (YAML™) revision 1.2.2" [Online]. Available: https://yaml.org/spec/1.2.2/. [Accessed: May 24, 2022]

[8] C. R. Pereira, "Managing modules with NPM," in *Building APIs with Node.js*, Berkeley, CA: Apress, 2016, pp. 9–13 [Online]. Available: https://doi.org/10.1007/978-1-4842-2442-7_3

[9] B. Varanasi and S. Belida, "Maven project basics," in *Introducing maven*, Berkeley, CA: Apress, 2014, pp. 23–36 [Online]. Available: https://doi.org/10.1007/978-1-4842-0841-0_4

[10] N. Adermann and J. Boggiano, "Composer." [Online]. Available: https://getcomposer.org/. [Accessed: May 24, 2022]

[11] R. Vemula, “Configuring a continuous build with travis CI,” in *Real-time web application development : With ASP.NET core, SignalR, docker, and azure*, Berkeley, CA: Apress, 2017, pp. 489–508 [Online]. Available: https://doi.org/10.1007/978-1-4842-3270-5_13

[12] P. Heller, *Automating Workflows with GitHub Actions: Automate software development workflows and seamlessly deploy your applications using GitHub Actions*. Packt Publishing, 2021.

[13] M. S. Arefeen and M. Schiller, “Continuous integration using gitlab,” *Undergraduate Research in Natural and Clinical Science and Technology Journal*, vol. 3, pp. 1–6, 2019.

[14] B. Burns, J. Beda, and K. Hightower, “Kubernetes: Up and Running,” p. 277.

[15] K. Jangla, “Docker compose,” in *Accelerating development velocity using docker: Docker across microservices*, Berkeley, CA: Apress, 2018, pp. 77–98 [Online]. Available: https://doi.org/10.1007/978-1-4842-3936-0_6

[16] F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, and D. Vrgoč, “Foundations of JSON Schema,” in *Proceedings of the 25th International Conference on World Wide Web*, Apr. 2016, pp. 263–273, doi: 10.1145/2872427.2883029 [Online]. Available: https://doi.org/10.1145/2872427.2883029. [Accessed: May 24, 2022]

[17] W3C, “W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures.” [Online]. Available: https://www.w3.org/TR/xmlschema11-1/. [Accessed: Jan. 02, 2022]

[18] “Webpack.” JS Foundation [Online]. Available: https://webpack.js.org/. [Accessed: May 24, 2022]

[19] C. Macrae, *Vue.js: Up and Running: Building Accessible and Performant Web Apps*, 1st edition. Sebastopol, California: O’Reilly Media, 2018.

[20] A. Martelli, A. Ravenscroft, and S. Holden, *Distributing Extensions and Programs*. [Online]. Available: https://learning.oreilly.com/library/view/python-in-a/9781491913833/ch25.html. [Accessed: May 24, 2022]

[21] “The Dhall configuration language.” Dhall [Online]. Available: https://dhall-lang.org/. [Accessed: Jan. 01, 2022]

[22] “CUE.” Cue Language Community [Online]. Available: https://cuelang.org/. [Accessed: Jan. 01, 2022]

[23] “Jsonnet - The Data Templating Language.” Code owned by Google [Online]. Available: https://jsonnet.org/. [Accessed: Jan. 01, 2022]

[24] B. Beyer, N. R. Murphy, D. K. Rensin, K. Kawahara, and S. Thorne, *The Site Reliability Workbook: Practical Ways to Implement SRE*. "O’Reilly Media, Inc.", 2018.

[25] “Nix.” Nix/Nixpkgs/NixOS, May 2022 [Online]. Available: https://github.com/NixOS/nix. [Accessed: May 01, 2022]

[26] “NixOps.” Nix/Nixpkgs/NixOS, May 2022 [Online]. Available: https://github.com/NixOS/nixops. [Accessed: May 24, 2022]

[27] “Gollum – A git-based Wiki.” Gollum, Jan. 2022 [Online]. Available: https://github.com/gollum/gollum. [Accessed: Jan. 04, 2022]

[28] James Turnbull, *The Terraform Book*. 2016 [Online]. Available: https://learning.oreilly.com/library/view/the-terraform-book/9780988820258/. [Accessed: May 24, 2022]

[29] M. Taylor and S. Vargo, *Learning Chef*. 2014 [Online]. Available: https://learning.oreilly.com/library/view/learning-chef/9781491945087/. [Accessed: May 24, 2022]

[30] Unit 42, “Highlights From the Unit 42 Cloud Threat Report, 2H 2021,” *Unit42*. Sep. 2021 [Online]. Available: https://unit42.paloaltonetworks.com/cloud-threat-report-2h-2021/. [Accessed: Jan. 04, 2022]

[31] V. Tzvetkov and J. M. Rinaudo, “Integrating AWS CloudFormation security tests with AWS Security Hub and AWS CodeBuild reports,” *Amazon Web Services*. Sep. 2020 [Online]. Available: https://aws.amazon.com/blogs/security/integrating-aws-cloudformation-security-tests-with-aws-security-hub-and-aws-codebuild-reports/. [Accessed: Jan. 04, 2022]

[32] “Nickel.” Tweag, May 2022 [Online]. Available: https://github.com/tweag/nickel. [Accessed: May 24, 2022]

[33] Y. Hamdaoui, “RFC-001 Overriding,” Tweag, May 2022 [Online]. Available: https://github.com/tweag/nickel/blob/f0dc86bd35241ddcb8e9d8482bdc2724ac1b2b19/rfcs/001-overriding.md. [Accessed: May 24, 2022]

[34] J. G. Siek and W. Taha, “Gradual Typing for Functional Languages,” p. 12, 2006.

[35] P. Wadler and R. B. Findler, “Well-Typed Programs Can’t Be Blamed,” in *Programming Languages and Systems*, vol. 5502, G. Castagna, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 1–16 [Online]. Available: http://link.springer.com/10.1007/978-3-642-00590-9_1. [Accessed: May 24, 2022]

[36] J. des Rivieres and J. Wiegand, “Eclipse: A platform for integrating development tools. IBM Systems Journal, 43(2), 371-383,” *IBM Systems Journal*, vol. 43, pp. 371–383, Jan. 2004, doi: 10.1147/sj.432.0371.

[37] E. Burnette, *Eclipse IDE: Pocket guide*, 1st ed. Sebastopol, CA: O’Reilly, 2005.

[38] “IntelliJ IDEA: The Capable & Ergonomic Java IDE by JetBrains,” *JetBrains*. [Online]. Available: https://www.jetbrains.com/idea/. [Accessed: May 01, 2022]

[39] “Visual Studio: IDE and Code Editor for Software Developers and Teams.” [Online]. Available: https://visualstudio.microsoft.com/. [Accessed: May 01, 2022]

[40] Moresmau, “EclipseFP The Haskell plug-in for Eclipse.” [Online]. Available: https://eclipsefp.github.io/index.html. [Accessed: May 01, 2022]

[41] “IntelliJ-Haskell - IntelliJ IDEs Plugin | Marketplace,” *JetBrains Marketplace*. [Online]. Available: https://plugins.jetbrains.com/plugin/8258-intellij-haskell. [Accessed: May 01, 2022]

[42] “HaskForce - The Haskell plugin for IntelliJ IDEA.” [Online]. Available: https://haskforce.com/. [Accessed: Feb. 19, 2022]

[43] “Haskell for Mac IDE — Learn Functional Programming with Haskell.” [Online]. Available: http://haskellformac.com/. [Accessed: Dec. 28, 2021]

[44] Y. Cébron, “Custom Language Support,” *IntelliJ Platform Plugin SDK Help*. [Online]. Available: https://plugins.jetbrains.com/docs/intellij/custom-language-support.html. [Accessed: Feb. 20, 2022]

[45] S. Keidel, W. Pfeiffer, and S. Erdweg, “The IDE portability problem and its solution in Monto,” in *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, Oct. 2016, pp. 152–162, doi: 10.1145/2997364.2997368 [Online]. Available: https://doi.org/10.1145/2997364.2997368. [Accessed: Dec. 21, 2021]

[46] A. M. Sloane, M. Roberts, S. Buckley, and S. Muscat, “Monto: A Disintegrated Development Environment,” in *Software Language Engineering*, 2014, pp. 211–220, doi: 10.1007/978-3-319-11245-9_12.

[47] C. Lattner and V. Adve, “LLVM: A compilation framework for lifelong program analysis amp; transformation,” in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, Mar. 2004, pp. 75–86, doi: 10.1109/CGO.2004.1281665.

[48] C. Lattner, “LLVM and Clang: Next Generation Compiler Technology,” vol. BSDCan, p. 33, May 2008 [Online]. Available: https://llvm.org/pubs/2008-05-17-BSDCan-LLVMIntro.pdf

[49] N. D. Matsakis and F. S. Klock, “The rust language,” *ACM SIGAda Ada Letters*, vol. 34, no. 3, pp. 103–104, Oct. 2014, doi: 10.1145/2692956.2663188. [Online]. Available: https://doi.org/10.1145/2692956.2663188. [Accessed: May 01, 2022]

[50] NVIDIA, “CUDA Compiler Driver NVCC.” May 2022 [Online]. Available: https://docs.nvidia.com/cuda/pdf/CUDA_Compiler_Driver_NVCC.pdf

[51] P. Hintjens, *ZeroMQ: Messaging for Many Applications*. "O’Reilly Media, Inc.", 2013.

[52] OCaml Community, “OCaml - Merlin,” *An editor service that provides advanced IDE features for OCaml*. [Online]. Available: https://ocaml.github.io/merlin/. [Accessed: Feb. 20, 2022]

[53] F. Bour, T. Refis, and G. Scherer, “Merlin: A Language Server for OCaml (Experience Report),” *Proceedings of the ACM on Programming Languages*, vol. 2, no. ICFP, pp. 1–15, Jul. 2018, doi: 10.1145/3236798. [Online]. Available: http://arxiv.org/abs/1807.06702. [Accessed: Nov. 12, 2021]

[54] S. Vaughan-Nichols, "Protocol aims to be a programming standard," *ZDNet*. [Online]. Available: https://www.zdnet.com/article/open-source-microsoft-protocol-aims-to-be-a-programming-standard/. [Accessed: Dec. 28, 2021]

[55] Red Hat, "Red Hat, Codenvy and Microsoft Collaborate on Language Server Protocol." Jun. 2016 [Online]. Available: https://www.redhat.com/en/about/press-releases/red-hat-codenvy-and-microsoft-collaborate-language-server-protocol. [Accessed: May 22, 2022]

[56] Microsoft, "Language Servers." [Online]. Available: https://microsoft.github.io/language-server-protocol/implementors/servers/. [Accessed: May 01, 2022]

[57] "Bringing a great IDE experience to the Rust programming language." *rust-analyzer*. [Online]. Available: https://rust-analyzer.github.io/. [Accessed: Nov. 23, 2021]

[58] OCaml Community, "OCaml Language Server Protocol implementation." OCaml, Apr. 2022 [Online]. Available: https://github.com/ocaml/ocaml-lsp. [Accessed: May 01, 2022]

[59] N. Mitchell *et al.*, "Building an Integrated Development Environment (IDE) on top of a Build System: The tale of a Haskell IDE," in *IFL 2020: Proceedings of the 32nd Symposium on Implementation and Application of Functional Languages*, Sep. 2020, pp. 1–10, doi: 10.1145/3462172.3462180 [Online]. Available: https://dl.acm.org/doi/10.1145/3462172.3462180. [Accessed: Nov. 12, 2021]

[60] I. Wechsung, "The Frege Programming Language (Draft)," p. 124 [Online]. Available: http://web.mit.edu/frege-lang_v3.24/Language.pdf

[61] T. Gagnaux and D. Konig, "Developing a Minimal Language Server for the Frege Programming Language: An Experience Report," p. 7.

[62] E. Dolstra, M. de Jonge, and E. Visser, "Nix: A Safe and Policy-Free System for Software Deployment," p. 14, 2004.

[63] "Rnix-lsp." Nix community projects, Apr. 2022 [Online]. Available: https://github.com/nix-community/rnix-lsp. [Accessed: May 01, 2022]

[64] D. Beyer and M. E. Keremoglu, "CPAchecker: A Tool for Configurable Software Verification," *arXiv:0902.0019 [cs]*, Jan. 2009 [Online]. Available: http://arxiv.org/abs/0902.0019. [Accessed: Feb. 24, 2022]

[65] A. Leimeister, "A Language Server and IDE Plugin for CPAchecker," Master's thesis, Ludwig-Maximilians-Universität München, 2020 [Online]. Available: https://www.sosy-lab.org/research/bsc/2020.Leimeister.A_Language_Server_and_IDE_Plugin_for_CPAchecker.pdf

[66] Z. Porkoláb, T. Brunner, D. Krupp, and M. Csordás, "Codecompass: An open software comprehension framework for industrial usage," in *Proceedings of the 26th Conference on Program Comprehension*, May 2018, pp. 361–369, doi: 10.1145/3196321.3197546 [Online]. Available: https://doi.org/10.1145/3196321.3197546. [Accessed: Feb. 25, 2022]

[67] H. Bünder and H. Kuchen, “Towards Multi-editor Support for Domain-Specific Languages Utilizing the Language Server Protocol,” in *Model-Driven Engineering and Software Development*, 2020, pp. 225–245, doi: 10.1007/978-3-030-37873-8_10.

[68] “Rnix-parser.” Nix community projects, May 2022 [Online]. Available: https://github.com/nix-community/rnix-parser. [Accessed: May 01, 2022]

[69] D. Beyer, G. Dresler, and P. Wendler, “Software Verification in the Google App-Engine Cloud,” in *Computer Aided Verification*, vol. 8559, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, A. Kobsa, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, D. Terzopoulos, D. Tygar, G. Weikum, A. Biere, and R. Bloem, Eds. Cham: Springer International Publishing, 2014, pp. 327–333 [Online]. Available: http://link.springer.com/10.1007/978-3-319-08867-9_21. [Accessed: Feb. 24, 2022]

[70] D. Beyer, “CPAchecker In the VerifierCloud.” Software Systems Lab LMU [Online]. Available: https://vcloud.sosy-lab.org/cpachecker/webclient/help/. [Accessed: Feb. 24, 2022]

[71] M. Mészáros, M. Cserép, and A. Fekete, “Delivering comprehension features into source code editors through LSP,” in *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, May 2019, pp. 1581–1586, doi: 10.23919/MIPRO.2019.8756695.

[72] L. Bettini and S. Efftinge, *Implementing domain-specific languages with Xtext and Xtend: Learn how to implement a DSL with Xtext and Xtend using easy-to-understand examples and best practices*, Second edition. Birmingham Mumbai: Packt Publishing, 2016.

[73] L. C. L. Kats and E. Visser, “The spoofax language workbench: Rules for declarative specification of languages and IDEs,” in *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, Oct. 2010, pp. 444–463, doi: 10.1145/1869459.1869497 [Online]. Available: https://doi.org/10.1145/1869459.1869497. [Accessed: Dec. 21, 2021]

[74] V. Pech, “JetBrains MPS: Why Modern Language Workbenches Matter,” in *Domain-Specific Languages in Practice: With JetBrains MPS*, A. Bucchiarone, A. Cicchetti, F. Ciccozzi, and A. Pierantonio, Eds. Cham: Springer International Publishing, 2021, pp. 1–22 [Online]. Available: https://doi.org/10.1007/978-3-030-73758-0_1. [Accessed: May 08, 2022]

[75] “Monaco Editor.” Microsoft, May 2022 [Online]. Available: https://github.com/microsoft/monaco-editor. [Accessed: May 01, 2022]

[76] L. Vogel and M. Milinkovich, “Eclipse rich client platform. Vogella series, lars vogel (2015).”

[77] R. Rodriguez-Echeverria, J. L. C. Izquierdo, M. Wimmer, and J. Cabot, "Towards a Language Server Protocol Infrastructure for Graphical Modeling," in *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, Oct. 2018, pp. 370–380, doi: 10.1145/3239372.3239383 [Online]. Available: https://doi.org/10.1145/3239372.3239383. [Accessed: Mar. 01, 2022]

[78] J. K. Rask, F. P. Madsen, N. Battle, H. D. Macedo, and P. G. Larsen, "The Specification Language Server Protocol: A Proposal for Standardised LSP Extensions," *Electronic Proceedings in Theoretical Computer Science*, vol. 338, pp. 3–18, Aug. 2021, doi: 10.4204/EPTCS.338.3. [Online]. Available: http://arxiv.org/abs/2108.02961. [Accessed: Feb. 15, 2022]

[79] J. Rask and F. Madsen, "Decoupling of Core Analysis Support for Specification Languages from User Interfaces in Integrated Development Environments," Master's thesis, Aarhus University, 2021.

[80] Scala Center and Jetbrains, "Build Server Protocol · Protocol for IDEs and build tools to communicate about compile, run, test, debug and more." [Online]. Available: https://build-server-protocol.github.io/. [Accessed: May 01, 2022]

[81] "Debug Adapter Protocol," *Official page for Debug Adapter Protocol*. [Online]. Available: https://microsoft.github.io/debug-adapter-protocol/implementors/adapters/. [Accessed: Mar. 03, 2022]

[82] P. Jeanjean, B. Combemale, and O. Barais, "IDE as Code: Reifying Language Protocols as First-Class Citizens," in *14th Innovations in Software Engineering Conference (formerly known as India Software Engineering Conference)*, Feb. 2021, pp. 1–5, doi: 10.1145/3452383.3452406 [Online]. Available: https://dl.acm.org/doi/10.1145/3452383.3452406. [Accessed: Mar. 03, 2022]