Contents

Chapter 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.

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

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, 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

Chapter 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

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 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.

[1] https://code.visualstudio.com/
[2] https://github.com/microsoft/language-server-protocol/
[3] https://microsoft.github.io/language-server-protocol/specifications/specification-current/

2.1.1 JSON-RPC

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

The protocol specifies the general format of messages exchanges as well as different kinds of messages. The following snippet lst. 2.1 shows the schema for request messages.

Listing 2.1 JSON-RPC Request

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

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

Responses, as shown in lst. 2.2, have to be sent by servers answering to any request. Any result or error of an operation is explicitly encoded in the response. Errors are represented as objects specifying the error kind using an error `code` and provide a human-readable descriptive `message` as well as optionally any procedure defined `data`.

Listing 2.2 JSON-RPC Response and Error

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

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

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

2.1.2 Commands and Notifications

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

2.1.2.1 File Notification

2.1.2.1.1 Diagnostics

2.1.2.2 Hover

2.1.2.3 Completion

2.1.2.4 Go-To-*

2.1.2.5 Symbols

2.1.2.6 code lenses

2.1.3 Shortcomings

2.2 Configuration programming languages

Nickel (**nickel?**), the language targeted by the language server detailed in this thesis, defines itself as "configuration language" used to automize the generation of static configuration files.

Static configuration languages such as XML(**xml?**), JSON(**json?**), or YAML(**yaml?**) are language specifications defining how to textually represent structural data used to configure parameters of a system[4]. Applications of configuration languages are ubiquitous especially in the vicinity of software development. While XML and JSON are often used by package managers (**composer?**), YAML is a popular choice for complex configurations such as CI/CD pipelines (**gitlab-runner?**) or machine configurations in software defined networks such as Kubernetes and docker compose.

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

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

These qualities require an evaluated language. In fact, some applications make heavy use of config files written in the native programming language which gives

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

them access to language features and existing analysis tools. Examples include JavaScript frameworks such as webpack (**webpack?**) or Vue (**vue?**) and python package management using `setuptools`(**setuptools?**).

Despite this, not all languages serve as a configuration language, e.g. compiled languages and some domains require language agnostic formats. For particularly complex products, both language independence and advanced features are desirable. Alternatively to generating configurations using high level languages, this demand is addressed by more domain specific languages. Dhall (**dhall?**), Cue (**cue?**) or jsonnet (**jsonnet?**) are such domain specific languages (DSL), that offer varying support for string interpolation, (strict) typing, functions and validation.

2.2.1 Infrastructure as Code

A prime example for the application of configuration languages are IaaS[5] products. These solutions offer great flexibility with regard to resource provision (computing, storage, load balancing, etc.), network setup and scaling of (virtual) servers. Although the primary interaction with those systems is imperative, maintaining entire applications' or company's environments manually comes with obvious drawbacks.

Changing and undoing changes to existing networks requires intricate knowledge about its topology which in turn has to be meticulously documented. 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*) serves the DevOps principles. 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.

As a notable instance, the Nix(**nix?**) ecosystem even goes as far as enabling declarative system and service configuration using NixOps(**nixops?**).

To get an idea of how this would look like, lst. 2.3 shows the configuration for a deployment of the Git based wiki server Gollum(**gollum?**) 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.

[5] Infrastructure as a Service

Listing 2.3 Example NixOps deployment to AWS

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

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

  reverseproxy =
    { config, pkgs, nodes, ... }:
    let
      gollumPort = nodes.gollum.config.services.gollum.port;
    in
    {
      deployment.ec2.instanceType = "t1.medium";
      services.nginx = {
        enable = true;
        virtualHosts."wiki.example.net".locations."/" = {
          proxyPass = "http://gollum:${toString gollumPort}";
        };
      };
      networking.firewall.allowedTCPPorts = [ 80 ];
    };
}
```

Similarly, tools like Terraform(**terraform?**), or Chef(**chef?**) use their own DSLs and integrate with most major cloud providers. The popularity of these products[6], beyond all, highlights the importance of expressive configuration formats and their industry value.

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

Finally, descriptive data formats for cloud configurations allow mitigating security risks through static analysis. Yet, as recently as spring 2020 and still more than a year later dossiers of Palo Alto Networks' security department Unit 42 (**pa2020H1?**) show that a majority of public projects uses insecure configurations. This suggests that techniques(**aws-cloud-formation-security-tests?**) to automatically check templates are not actively employed, and points out the importance of evaluated configuration languages which can implement passive approaches to security analysis.

2.2.2 Nickel

2.2.2.1 Gradual typing

2.2.2.1.1 Row types

2.2.2.2 Contracts

In addition to a static type-system Nickel integrates a contract system akin what is described in (**cant-be-blamed?**). 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. 2.4.

Listing 2.4 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. 2.5

Listing 2.5 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. 2.6) using custom blame messages. Nickel is able to point to the exact value violating a contract as well as the contract in question.

Listing 2.6 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
```

2.2.2.3 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.

2.2.2.3.1 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. 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 HashMaps, uniquely associating an identifier with a sub-node.

These data types constitute a static subset of Nickel which allows writing JSON compatible expressions as shown in lst. 2.7.

Listing 2.7 Example of a static Nickel expression

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

Building on that Nickel also supports variables and functions.

2.2.2.3.2 Identifiers The inclusion of Variables to the language, implies some sort 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 2.8 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>
```

2.2.2.3.3 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. 2.8.

Listing 2.9 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. 2.9. Function argument name binding therefore looks the same as in `let` bindings.

2.2.2.3.4 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. 2.10.

Listing 2.10 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. 2.11 translates to the AST in fig. 2.1.

Listing 2.11 Example of a typed expression

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

2.2.2.3.5 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. 2.12

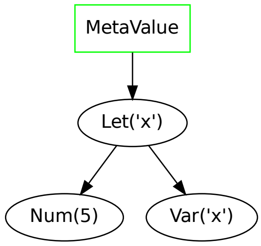

Figure 2.1: AST of typed expression

Listing 2.12 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. 2.13 with the rest of the tree omitted.

Listing 2.13 Nickel static access

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

2.2.2.3.6 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. 2.14 uses the shorthand syntax which resolves to the semantically equivalent record defined in lst. 2.15

Listing 2.14 Nickel record defined using shorthand

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

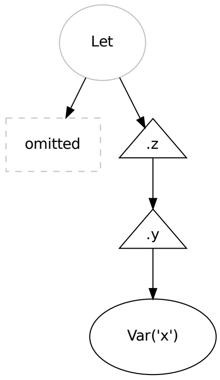

Figure 2.2: AST of typed expression

Listing 2.15 Nickel record defined explicitly

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

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

Chapter 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. This chapter will survey a varied range of popular language servers, compare common capabilities, and implementation approaches. Additionally, this part aims to recognize alternative approaches to the LSP, in the form of legacy protocols, extensible development platforms LSP extensions and the emerging Language Server Index Format.

3.1 Previous Approaches

3.1.1 IDEs

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 (**intellij-supported-languages?**). 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(**eclipse-a-platform?**,eclipse-www), IntelliJ(**intelliJ?**), and Visual Studio(**VisualStudio?**), 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 ((**jetbrains-all-products?**)). 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 (**haskell-ide-eclips?**), as well as independent IntelliJ plugins (**HaskForce?**). 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(**haskell-for-mac?**) 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.

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 (**custom-language-support?**), 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 (**intellij-comparison-eclipse?**). 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 IDE Abstraction

3.1.2.1 Monto

The authors of the Monto project(**monto-disintegrated?**) 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 (**llvm?**). Compiler frontends for different languages – e.g. Clang(**clang?**), Rustc(**rustc?**), NVCC(**nvcc?**),... – 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[zeromq] technology which was chosen because it is lightweight and available in manly languages (**monto-disintegrated?**) allowing to make use of existing language tools.

3.1.2.2 Merlin

The Merlin tool (**merlin-website?**) 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 (**merlin?**).

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

3.2.1 Considerable dimensions

3.2.1.1 Language Complexity

3.2.1.2 LSP compliance

3.2.1.3 Features

3.2.1.4 File processing

3.2.1.4.1 Incremental

3.2.1.4.2 Full

3.2.2 Comparative Projects

3.2.3 Honorable mentions

3.3 Alternative approaches

3.3.1 Platform plugins

3.3.2 Legacy protocols

3.3.3 LSP Extensions

3.3.4 LSIF

Chapter 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 3.2.1 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. 3.2.1, 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. 3.2.1 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 enoigh eager analysis prioritizing a more straight forward implementation over optimized performance.

4.3 High-Level Architecture

This section describes the high-level architecture of NLS. The entity diagram depicted in fig. 4.1 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 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 6.3 lays out a more detailed plan how this will be achieved.

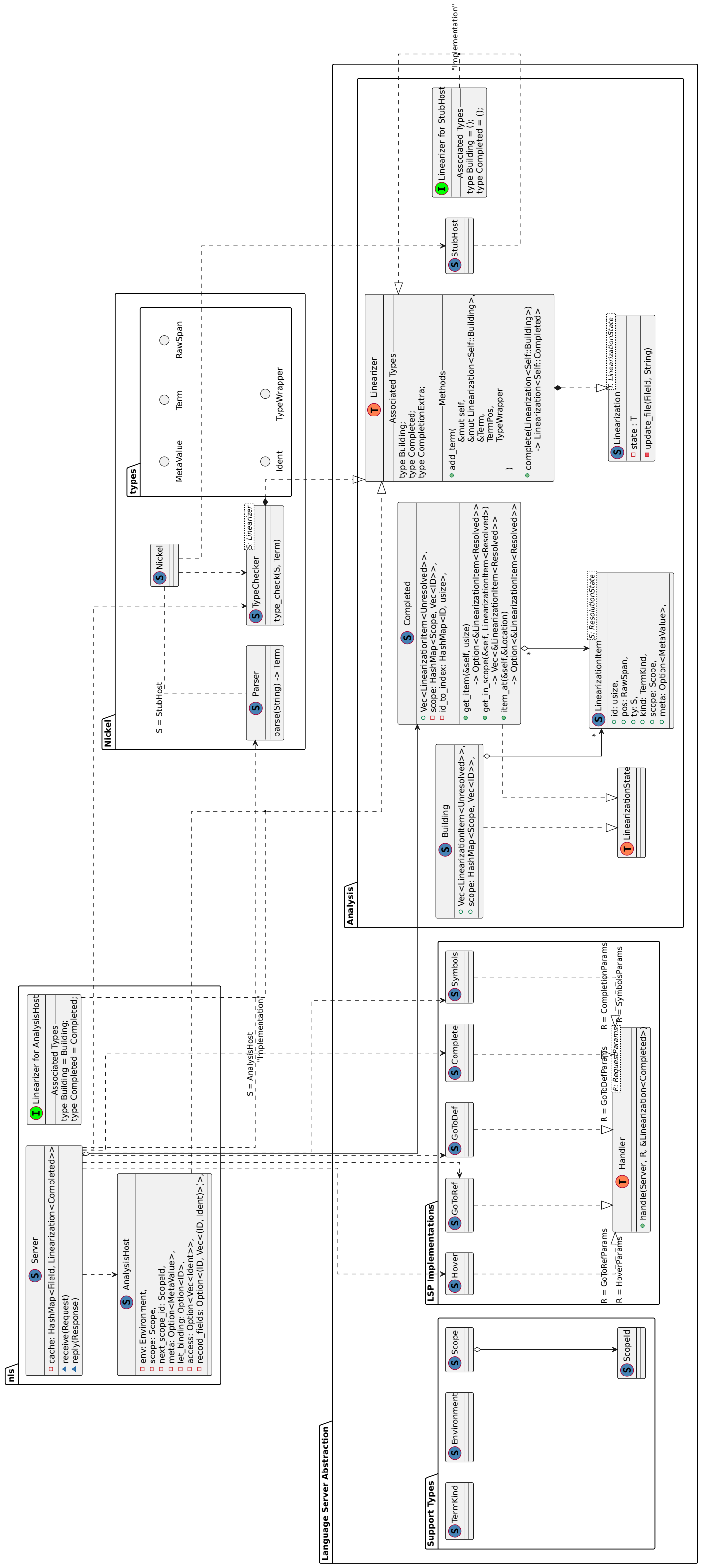

Figure 4.1: Entity Diagram showing the architecture of NLS, explicit dependency arrows omitted for legibility.

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. 2.2.2.3), such as information about references these can be deduced from the AST lazily, it would require the repeated traversal of arbitrarily large tree with an associated cost to performance. 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, the *type* of the node and its *kind* are tracked separately. The latter is used to represent a 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. Thanks to the generic interface, the adaptions to Nickel to integrate NLS are expected to have almost no influence on the runtime performance of the language in an optimized build.

NLS implements separate type-states for 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. 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

```rust
pub trait LinearizationState {}

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

Listing 4.3 Type Definition of Building state

```rust
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. Consequently, NLS faces the challenge of satisfying multiple goals

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. Since most methods of this implementation are `no-ops`, the compiler should be able to optimize away all `Linearizer` calls in release builds.

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 `ids` 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

```
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 inherently supports this logic. 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 in order to provide auto-completion of identifiers and list symbol names based on their 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).

```
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[^https://en.wikipedia.org/wiki/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
  }
}
```

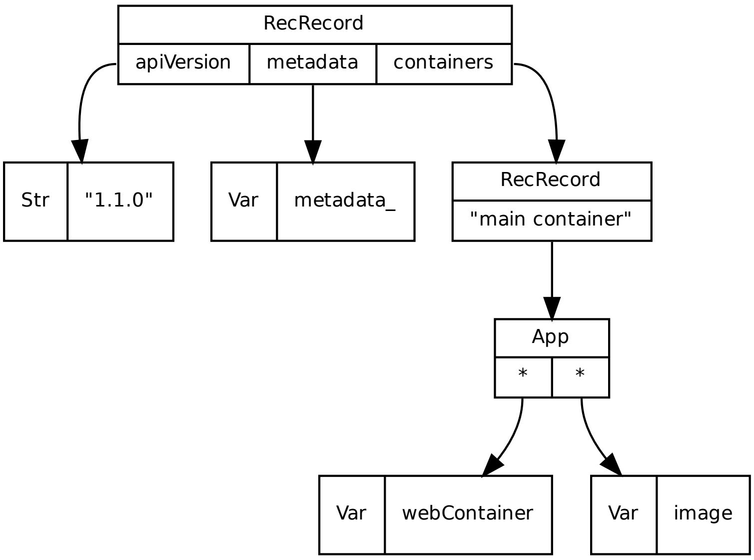

Figure 4.2: AST representation of a record

4.5.2.4.3 Records Section **??** introduced the AST representation of Records. As suggested by fig. 4.2, 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. 4.3) 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. 4.3).
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`. 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. 4.4

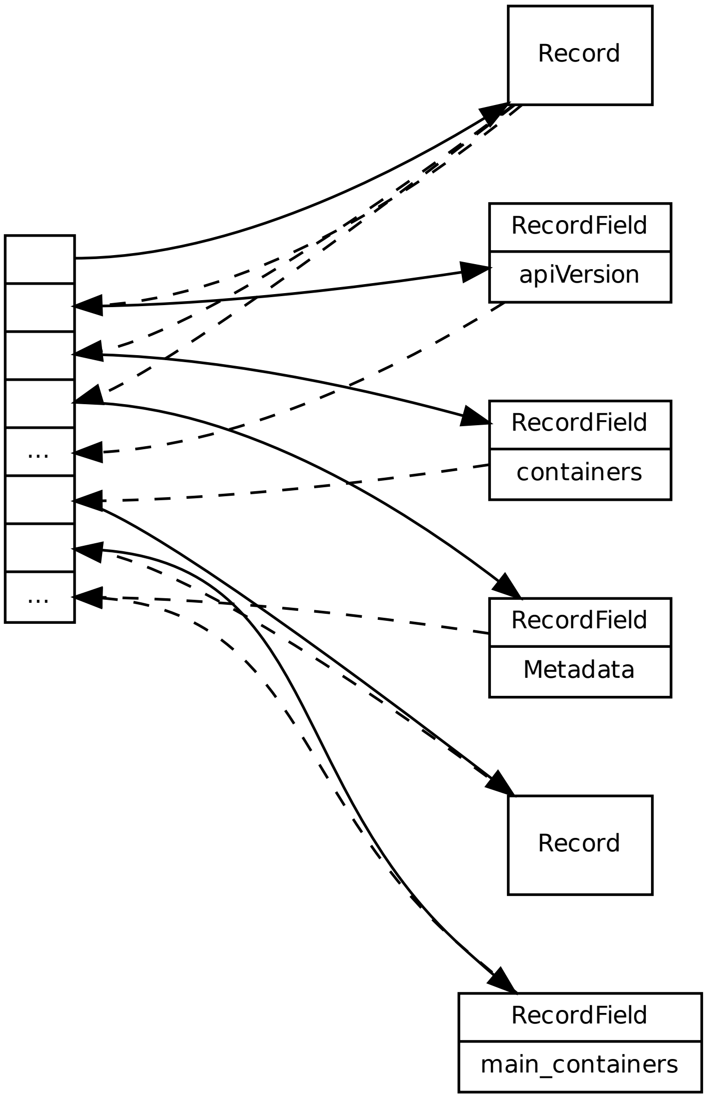

Figure 4.3: Linearization of a record

Listing 4.11 Example of a recursive record

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

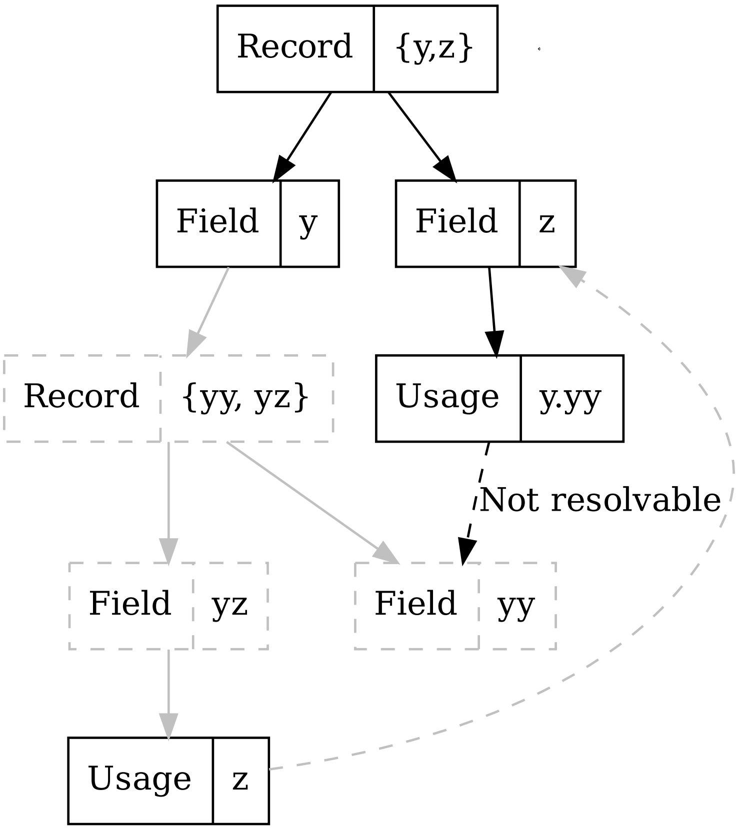

Figure 4.4: Example lock in recursive records. The field ‘y.yz‘ cannot be not be referenced at this point as the ‘y‘ branch has yet to be linearized

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 4.5 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`.

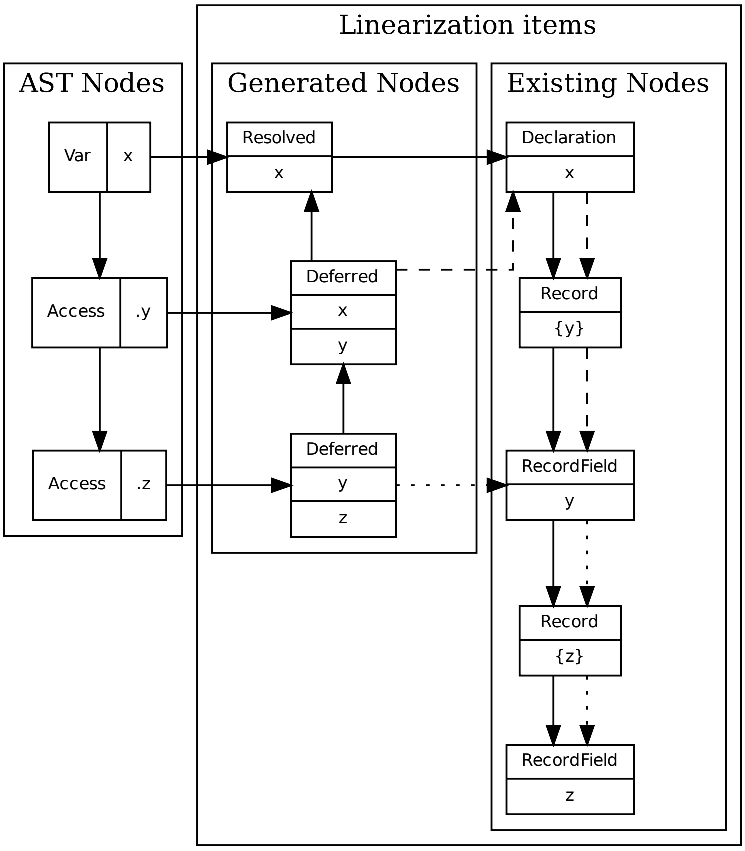

Figure 4.5: Depiction of generated usage nodes for record destructuring

4.5.2.4.6 Metadata In sec. 2.2.2.3.4 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 `id`s 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. 4.5, 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.

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

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
  }
}
```

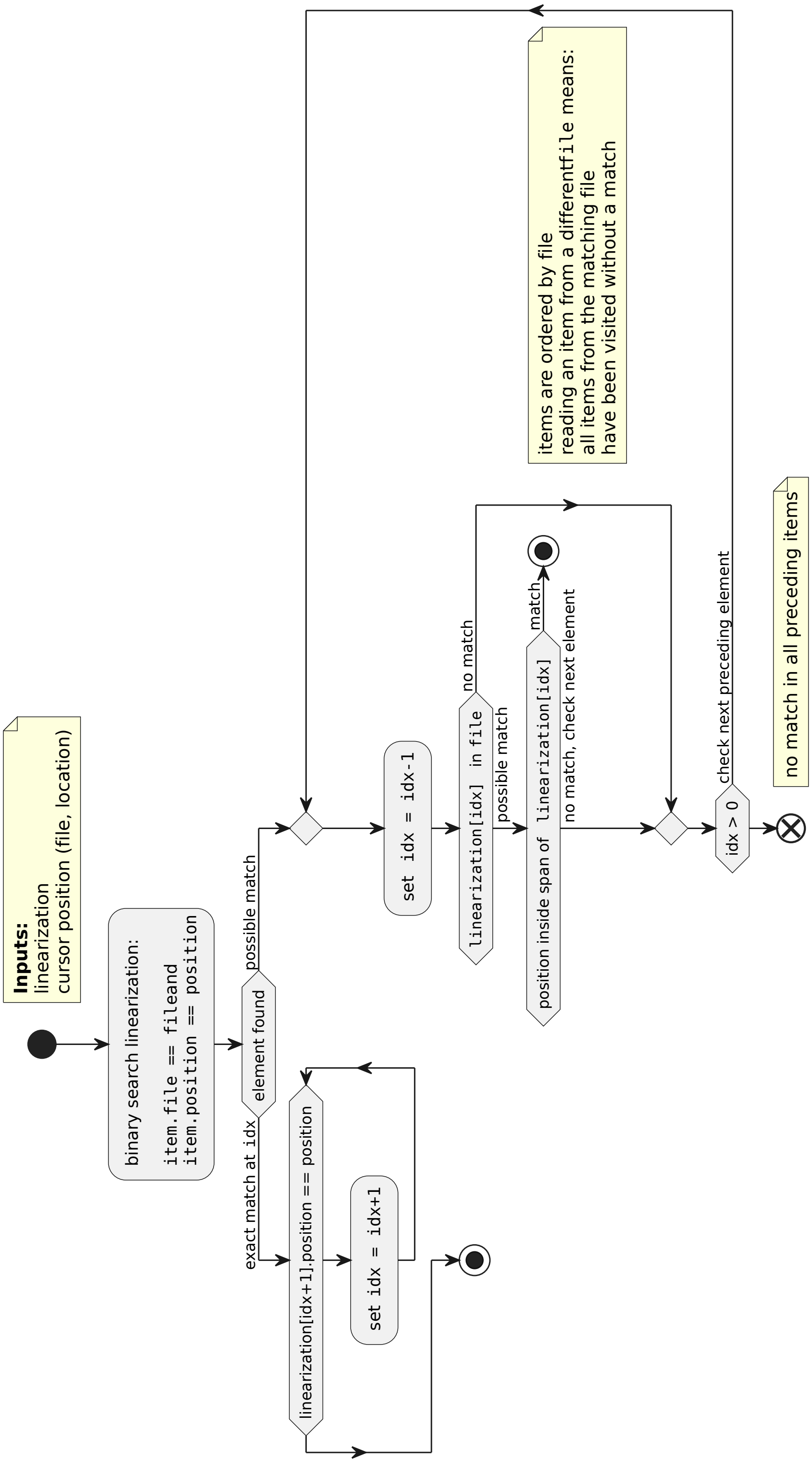

Figure 4.6: Activity diagram of item resolution by position

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

```rust
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. 4.1, 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.

Chapter 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 developers experience any help due to the use of the server.
Does NLS meet its users' expectations in terms of completeness and correctness and behavior? Being marketed as a Language Server, invokes certain expectations due to 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 return of the reply by 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. 5.2.2) which was conducted with the intent to gain qualitative opinions by users, as well as the tracing mechanism (sec. 5.2.3) for factual quantitative insights. Section 5.4 summarises 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 with experiences allows the implementation more practically and guide the further development by highlighting well executed, immature or missing features.

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. **??** an eager processing model was chosen over lazy analysis. It was hypothesized that analyzing Nickel source code eagerly allows to perform a single computation ahead of time instead of multiple partial ones. This way both analyzing and querying information could be implemented more efficiently. Moreover, it was assumed that eager computation would have a negligible impact on performance given the relatively small size of Nickel code bases in the current stage.

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 as for NLS both as an LSP implementation for Nickel (affecting the perceived maturity of Nickel) as well as a generic basis for other projects. On the other hand, all features are implemented on top of the same base (cf. sec. **??**). The survey should therefore show architectural deficits as well.

The quantitative study in contrast focuses on measurable performance. Similarly to the survey bases evaluation it should reveal insight for different features and tasks separately. An additional objective, in line with the definition of the performance metric in [#sec:metrics], is to show the influence of growing file sizes in practice.

5.2.2 Qualitative

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 independent of the experience, the survey consists of two parts – a pre-evaluation and final survey.

5.2.2.1 Pre-Evaluation

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. 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 a request to 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. As a preparation, they were asked to install the LSP. The workshop allowed participants unfamiliar with the Nickel language to use the language and experience NLS. Following the workshop, participants filled in a second survey which focused on particular experiences of every single feature. This evaluation focused on three main aspects. First, the general experience without weighing in expectations. The goal was to assess the extent to which the users were able to use the feature, since all usability metrics as discussed in [#sec:metrics] depend on the respective feature being available in the first place. In the same category are the items surveying the perceived performance and stability on a linear scale hinting at possible usability issues. The scales span from "Very slow response" to "Very quick response" and "Never Crashed" to "Always Crashed" respectively. Under the second aspect, the users were asked to explicitly reflect on their expectations in order to contribute to the usability metric. In the final part participants could describe their perceived shortcomings or questions or remarks.

5.2.3 Quantitative

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 occured 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, which could introduce a bias in the results.

5.3 Process

5.4 Results

5.4.1 Qualitative

5.4.2 Quantitative

Chapter 6

Discussion

6.1 Project results

6.2 Project shortcomings

6.3 Future Work