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 one of which being put forward by Microsoft - the Language Server Protocol [ref] which is discussed in greater detail in sec. **??**. 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, all LSP-capable editors can connect to the same LSP implementation. Moreover, LSP client implementations are independent of the servers. Hence, 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 choice of language to implement language servers in lies with the developer. In effect, it is possible for language developers to integrate essential parts of the existing language implementation for a language server. By now the LSP has become the most popular choice for cross-platform language tooling with implementations [langservers and microsoft] for all major and many smaller languages.

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

Problem Definition

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

Moreover, most projects do not formally evaluate the Language Server on even basic requirements. Naïvely, that is, the server should be *performant* enough not to slow down the developer, it should offer *useful* information and capabilities and of course be *correct* as well as *complete*.

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[1]:

RQ.1 How to develop a language server for a new language with the abovementioned requirements in mind?
RQ.2 How can we assess the implementation both quantitatively based on performance measures and qualitatively based on user satisfaction?
RQ.3 Do the methods used to answer RQ.1 meet the expected requirements under the assessment developed in RQ.2?

Goals

The goal of this research is to describe a reusable approach for representing programs that can be used to query data to answer requests on the Language Server Protocol efficiently. The research is conducted on an implementation of the open source language Nickel[^https://nickel-lang.org] which provides the *Diagnostics*, *Jump to ** and *Hover* features as well as limited *Auto-Completion* and *Symbol resolution*. Although implemented for and with integration of the Nickel runtime, the 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.

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

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

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

Research Methodology

What are the scientific methods

Structure of the thesis

Background

This thesis illustrates an approach of implementing a language server for the Nickel language which communicates with its clients, i.e. editors, over the open Language Server Protocol (in the following abbreviated as *LSP*). The current chapter provides the background on the technological details of the project. As the work presented aims to be transferable to other languages using the same methods, this chapter will provide the means to distinguish the nickel specific implementation details.

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

Language Server Protocol

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

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

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

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

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

different kinds of messages. The following snippet lst. 0.1 shows the schema for request messages.

Listing 0.1 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. 0.2, 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 providing a human-readable descriptive `message` as well as optionally any procedure defined `data`.

Listing 0.2 JSON-RPC Response and Error

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

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

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

Commands and Notifications

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

File Notification

Diagnostics

Hover

Completion

Go-To-*

Symbols

code lenses

Shortcomings

Configuration programming languages

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

Static configuration languages such as XML(**xml?**), JSON(**json?**), or YAML(**yaml?**) are language specifications defining how to textually represent structural data used to configure parameters of a program[5]. 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 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?**),

[5] 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

Cue (**cue?**) or jsonnet (**jsonnet?**) are such domain specific languages (DSL), that offer varying support for string interpolation, (strict) typing, functions and validation.

Infrastructure as Code

A prime example for the application of configuration languages are IaaS[6] products. These solutions arise highly complex solutions with regard to resource provision (computing, storage, load balancing, etc.), network setup and scaling. Although the primary interaction with those systems is imperative, maintaining entire applications' or company's environments manually comes with obvious drawbacks.

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

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

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

To get an idea of how this would look like, lst. 0.3 shows the configuration for a deployment of the Git based wiki server Gollum(**gollum?**) behind a nginx reverseproxy on the AWS network. Although targeting AWS, Nix itself is platform-agnostic and NixOps supports different backends through various plugins. Configurations like this are abstractions over many manual steps and the Nix language employed in this example allows for even higher level turing-complete interaction with configurations.

Similarly, tools like Terraform(**terraform?**), or Chef(**chef?**) use their own DSLs and integrate with most major cloud providers. The popularity of these products[7], beyond all, highlights the importance of expressive configuration formats and their industry value.

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

[6] Infrastructure as a Service

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

out the importance of evaluated configuration languages which can implement passive approaches to security analysis.

Nickel

Gradual typing

Row types

Contracts

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.

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. 0.4.

Building on that Nickel also supports variables and functions which make up the majority of the AST.

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.

Let Bindings and Functions 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. 0.5.

Functions in Nickel are lambda expressions. A function with multiple arguments gets broken down into nested functions with a single argument for each argument of the source declaration as seen in lst. 0.6. Function argument name binding therefore looks the same as in `let` bindings.

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 meta-data such as documentation, default values and merge priority using a special syntax as displayed in lst. 0.7.

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. 0.8 translates to the AST in fig. 1. The green `MetaValue` box is a virtual node generated during parsing and not present in the untyped equivalent.

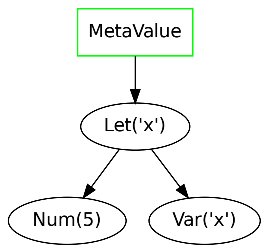

Figure 1: AST of typed expression

Nested Record Access Nickel supports the referencing of variables that are resolved during runtime. With records bound to a variable, a method to destruct records is required. Nickel restricts field names to strings,

The access of record fields is represented using a special set of AST nodes depending on whether the field name requires an evaluation in which case resolution is deferred to the evaluation pass. While the latter prevents static analysis of any deeper element by the LSP, `StaticAccess` 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 shows the representation of the static access perfomed in lst. 0.9 with the rest of the tree omitted.

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. 0.10 uses the shorthand syntax which resolves to the semantically equivalent record defined in lst. 0.11

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

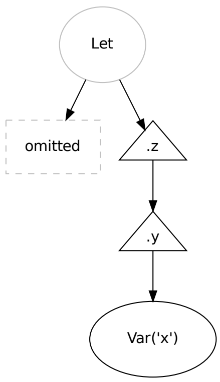

Figure 2: AST of typed expression

Listing 0.3 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 ];
    };
}
```

Listing 0.4 Example of a static Nickel expression

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

Listing 0.5 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>
```

Listing 0.6 Parsed representation of functions with multiple arguments

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

Listing 0.7 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
```

Listing 0.8 Example of a typed expression

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

Listing 0.9 Nickel static access

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

Listing 0.10 Nickel record using shorthand

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

Listing 0.11 Nickel record defined explicitly

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

Design implementation of NLS

This chapter contains a detailed guide through the various steps and components of the Nickel Language Server (NLS). Being written in the same language (Rust(**rust?**)) as the Nickel interpreter allows NLS to integrate existing components for language analysis. Complementary, NLS is tightly coupled to Nickel's syntax definition. Based on that sec. **??** will introduce the main datastructure underlying all higher level LSP interactions and how the AST described in sec. **??** is transformed into this form. Finally, in sec. **??** the implementation of current LSP features is discussed on the basis of the previously reviewed components.

Illustrative example

The example lst. 0.12 shows an illustrative high level configuration of a server. Throughout this chapter, different sections about the NSL implementation will refer back to this example.

Linearization

The focus of the NLS as presented in this work is to implement a working language server with a comprehensive feature set. Prioritizing a sound feature set, NLS takes an eager, non-incremental approach to code analysis, resolving all information at once for each code update (`didChange` and `didOpen` events), assuming that initial Nickel projects remain reasonably small. The analysis result is subsequently stored in a linear data structure with efficient access to elements. This data structure is referred to in the following as *linearization.* The term arises from the fact that the linearization is a transformation of the syntax tree into a linear structure which is presented in more detail in sec. **??**. The implementation distinguishes two separate states of the linearization. During its construction, the linearization will be in a *building* state, and is eventually post-processed yielding a *completed* state. The semantics of these states are defined in sec. **??**, while the post-processing is described separately in sec. **??**. Finally, sec. **??** explains how the linearization is accessed.

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 as well as state dependent auxiliary structures.

Closely related to nodes, `LinearizationItem`s maintain the position of their AST counterpart, as well as its type. Unlike in the AST, *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 distinguish between declarations of variables, records, record fields and variable usages as well as a wildcard kind for any other kind of structure, such as terminals control flow elements.

The aforementioned separation of linearization states got special attention. As the linearization process is integrated with the libraries underlying the Nickel interpreter, it had to be designed to cause minimal overhead during normal execution. Hence, the concrete implementation employs type-states(**typestate?**) to separate both states on a type level and defines generic interfaces that allow for context dependent implementations.

At its base the `Linearization` type is a transparent smart pointer(**deref-chapter?**; **smart-pointer-chapter?**) to the particular `LinearizationState` which holds state specific data. On top of that NLS defines a `Building` and `Completed` state.

The `Building` state represents a raw linearization. In particular that is a list of `LinearizationItems` of unresolved type ordered as they are created through a depth-first iteration of the AST. Note that new items are exclusively appended such that their `id` field is equal to the position at all time during this phase. Additionally, the `Building` state records all items for each scope in a separate mapping.

Once fully built, a `Building` instance is post-processed yielding a `Completed` linearization. While being defined similar to its origin, the structure is optimized for positional access, affecting the order of the `LinearizationItem`s and requiring an auxiliary mapping for efficient access to items by their `id`. Moreover, types of items in the `Completed` linearization will be resolved.

Type definitions of the `Linearization` as well as its type-states `Building` and `Completed` are listed in lsts. 0.13, 0.14, 0.15. Note that only the former is defined as part of the Nickel libraries, the latter are specific implementations for NLS.

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

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 to obstruct its development and maintain performance of the runtime.

To accommodate these goals NLS comprises three different parts as shown in fig. 3. The `Linearizer` trait acts as an interface between Nickel and the language server. NLS implements such a `Linearizer` specialized to Nickel which registers nodes and builds a final linearization. As Nickel's type checking implementation was adapted to pass AST nodes to the `Linearizer`. During normal operation the overhead induced by the `Linearizer` is minimized using a stub implementation of the trait.

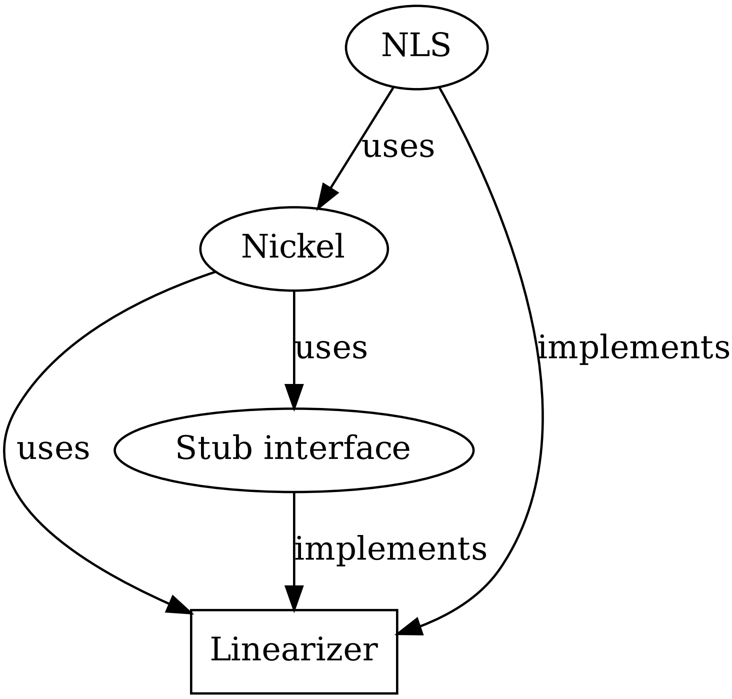

Figure 3: Interaction of Componenets

Usage Graph

At the core the linearization is a simple *linear* structure. Also, in the general case[8] the linearization is reordered in the post-processing step. This makes it impossible to encode relationships of nodes on a structural level. Yet, Nickel's support for name binding of variables, functions and in recursive records implies great a necessity for node-to-node relationships to be represented in a representation that aims to work with these relationships. On a higher level, tracking both definitions and usages of identifiers yields a directed graph.

[8] Except single primitive expressions

There are three main kids of vertices in such a graph. **Declarations** are nodes that introduce an identifier, and can be referred to by a set of nods. Referral is represented by **Usage** nodes which can either be bound to a declaration or unbound if no corresponding declaration is known. In practice Nickel distinguishes simple variable bindings from name binding through record fields which are resolved during the post-precessing. It also Integrates a **Record** and **RecordField** kinds to aid record destructuring.

During the linearization process this graphical model is recreated on the linear representation of the source. Hence, each `LinearizationItem` is associated with one of the aforementioned kinds, encoding its function in the usage graph.

The `TermKind` type is an enumeration of the discussed cases and defines the role of a `LinearizationItem` in the usage graph.

Variable bindings are linearized using the `Declaration` variant which holds the bound identifier as well as a list of `ID`s corresponding to its `Usage`s.

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

Record fields make for to most complicated kind. The `RecordField` kind augments the qualities of a `Declaration` representing an identifier, and tracking its `Usage`s, while also maintaining a link back to its parent `Record` as well as explicitly referencing the value represented.

Variable usages are further specified. `Usage`s that can not be mapped to a declaration are tagged `Unbound` or otherwise `Resolved` to the complementary `Declaration`
Record destructuring may require a late resolution as discussed in (**sed:variable-usage-and-static-record-access?**).

Other nodes of the AST that do not fit in a usage graph, are linearized as `Structure`.

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 for a local area

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. **??**.

For simplicity, NLS represents scopes by a prefix list of integers. Whenever a new lexical scope is entered, the list of the outer scope is extended by a unique identifier.

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.

Linearizer

The heart of the linearization the `Linearizer` trait as defined in lst. 0.17. The `Linearizer` lives in parallel to the `Linearization`. Its methods modify a shared reference to a `Building` `Linearization`.

`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 for a current 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 especially apparent with records where the field names part of the record node and as such are linearized with the record but have to be assigned there actual type separately.

`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. **??**).

```
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>>,
}
```

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. 0.18. 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.

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.

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. 0.19 Essentially, any of such nodes turns into a typed span as the remaining information tracked is the item's span and type checker provided type.

Declarations In case of `let` bindings or function arguments name binding is equally simple.

When the `Let` node is processed, the `Linearizer` generates `Declaration` items for each identifier contained. As discussed in sec. **??** the `Let` node may contain a name binding as well as pattern matches. The node's type supplied to the

`Linearizer` accords to the value and is therefore applied to the name binding only. Additionally, NLS updates its name register with the newly created `Declarations`.

The same process applies for argument names in function declarations.

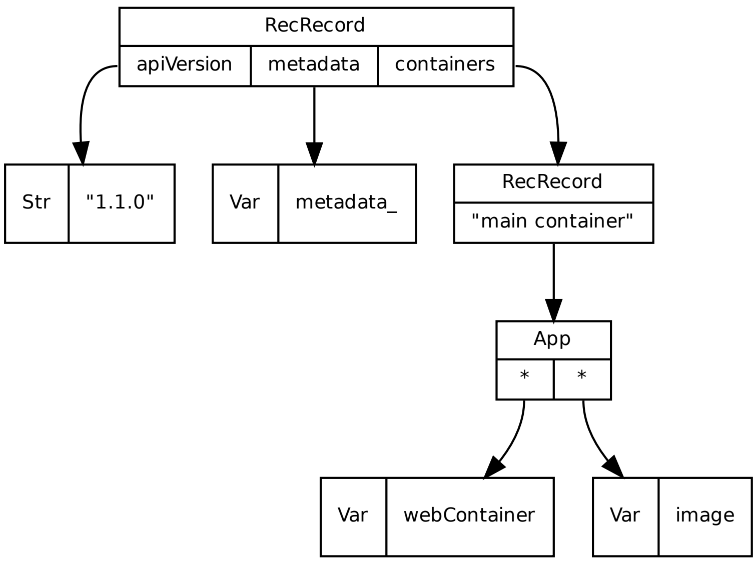

Figure 4: AST representation of a record

Records Linearizing records proves more difficult. In sec. **??** the AST representation of Records was discussed. As shown by fig. 4, Nickel does not have AST nodes dedicated to record fields. Instead, it associates field names with values as part of the `Record` node. For the language server on the other hand the record field is as important as its value, since it serves as name declaration. For that reason NLS distinguishes `Record` and `RecordField` as independent kinds of linearization items.

NLS has to create a separate item for the field and the value. That is to maintain similarity to the other binding types. It provides a specific and logical span to reference and allows the value to be of another kind, such as a variable usage like shown in the example. The language server is bound to process nodes individually. Therefore, it can not process record values at the same time as the outer record. Yet, 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. While, `RecordField` items are created while processing the record, they can not yet be connected to the value they represent, as the linearizer can not know the `id` of the latter. This is because the subtree of each of the fields can be arbitrary large causing an unknown amount of items, and hence intermediate `ids` to be added to the Linearization.

A summary of this can be seen for instance on the linearization of the previously discussed record in fig. 5. Here, record fields are linearized first, pointing to some following location. 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 used 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. 5).
4. The `scope` method will be called in the same order as the record fields appear. Using this fact, the `scope` method moves the data stored for the next evaluated field into the freshly generated `Linearizer`
5. **(In the sub-scope)** The `Linearizer` associates the `RecordField` item with the (now known) `id` of the field's value. The cached field data is invalidated such that this process only happens once for each field.

Variable Reference While name declaration can happen in several ways, the usage of a variable is always expressed as a `Var` node wrapping a referenced identifier. Registering a name usage is a multi-step process.

First, NLS tries to find the identifier in its scoped 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 the `Resolved Usage` item to the linearization and update the declaration's list of usages.

Variable Usage and Static Record Access Looking at the AST representation of record destructuring in fig. 2 shows that accessing inner records involves chains of unary operations *ending* with a reference to a variable binding. Each operation encodes one identifier, i.e. 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 reached. Whenever a variable usage is linearized, NLS checks the stack for latent destructors. If destructors are present, NLS adds `Usage` items for each element on the stack.

Note that record destructors can be used as values of record fields as well and thus refer to other fields of the same record. As the `Linearizer` processes the field values sequentially, it is possible that a usage references parts of the record that have not yet been processed making it unavailable for NLS to fully resolve.

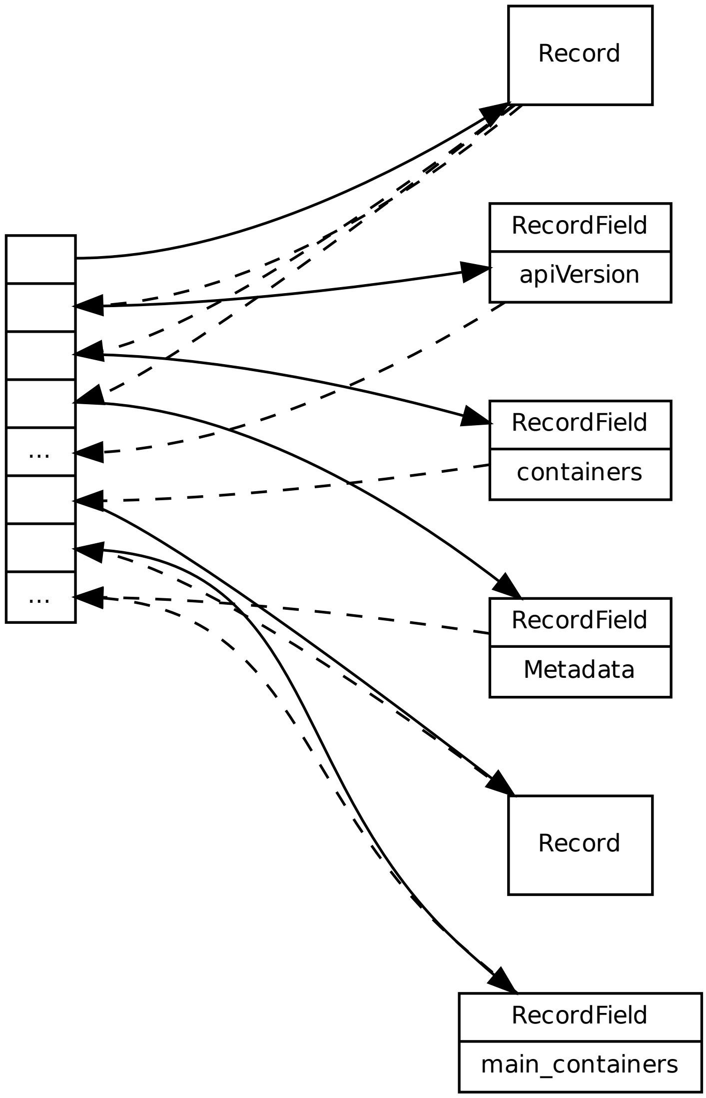

Figure 5: Linearization of a record

A visualization of this is provided in fig. 6 For this reason the `Usages` added to the linearization are marked as `Deferred` and will be fully resolved during the post-processing phase as documented in sec. **??**. In fig. 7 this is shown visually. The `Var` AST node is linearized as a `Resolved` usage node which points to the existing `Declaration` node for the identifier. Mind that this could be a `RecordField` too if referred to in a record. NLS linearized the trailing access nodes as `Deferred` nodes.

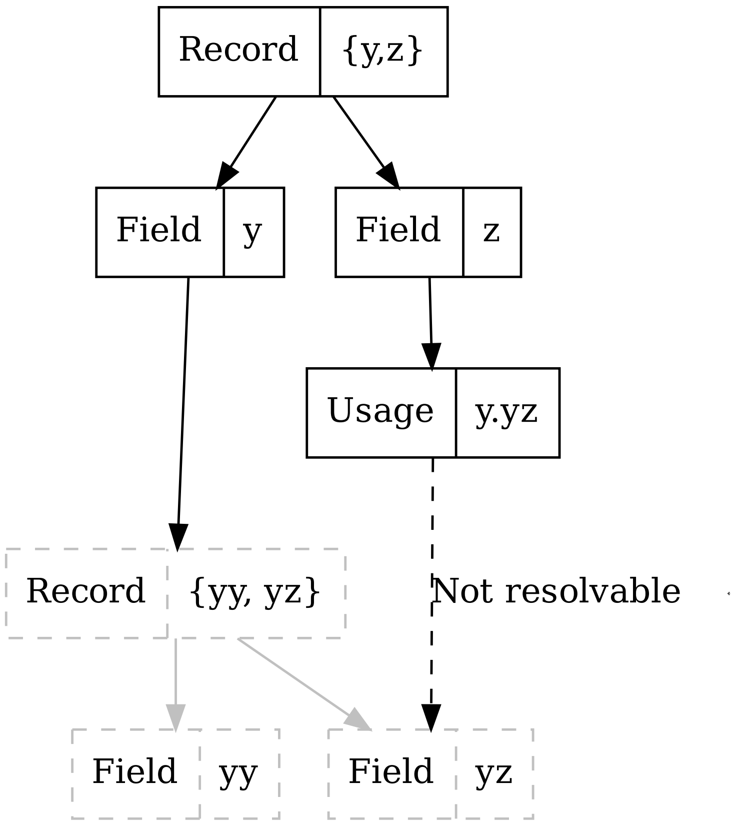

Figure 6: Example race condition in recursive records. The field ‘y.yz‘ cannot be not be referenced at this point as the ‘y‘ branch has yet to be linearized

Metadata In sec. **??** was shown that on the syntax level, metadata “wraps” the annotated value. Conversely, NLS encodes metadata in the `LinearizationItem` as metadata is 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.

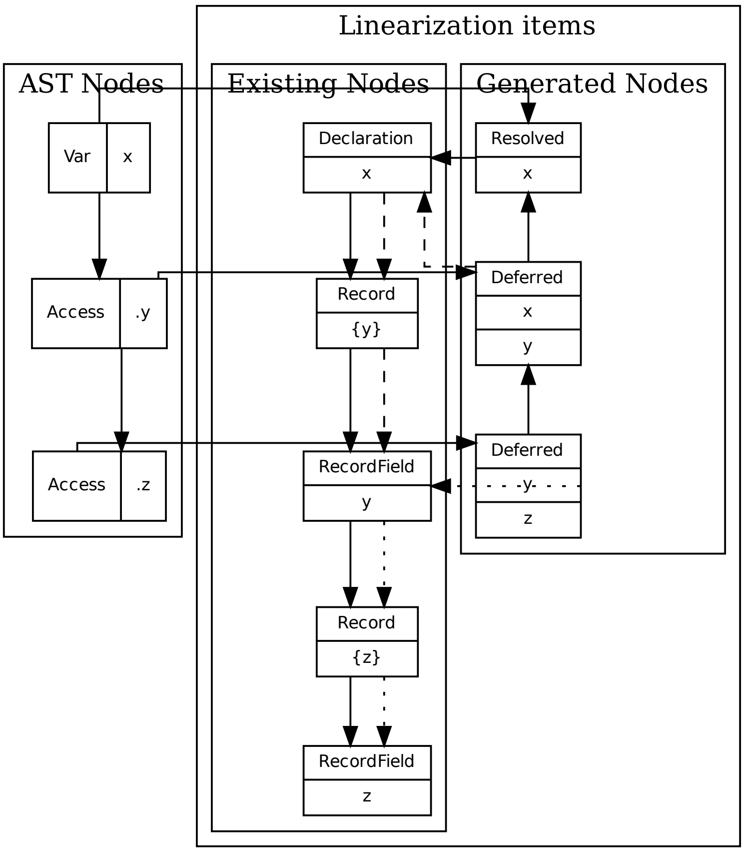

Figure 7: Depiction of generated usage nodes for record destructuring

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`.

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

Sorting

Since the linearization is performed in a preorder traversal, processing already happens in the order elements are defined physically. 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 a physical location and are thus placed at the range `[0..=0]` in the beginning of the file. Maintaining constant insertion performance and item-referencing require that the linearization is exclusively appended. Each of these cases, break the physical linearity of the linearization.

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 list indices.

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

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

Resolving types

As a necessity for type checking, Nickel generates type variables for any node of the AST which it hands down to the `Linearizer`. In order to provide meaningful information, the Language Server needs to derive concrete types from these variables. The required metadata needs to be provided by the type checker.

Resolving Elements

Resolving by position

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

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

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

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

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

The algorithm used is listed in lst. 0.21. 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`.

Resolving by ID

During the building process item IDs are equal to their index in the underlying List which allows for efficient access by ID. To allow similarly efficient access to nodes with using IDs a `Completed` linearization maintains a mapping of IDs to their corresponding index in the reordered array. A queried ID is first looked up in this mapping which yields an index from which the actual item is read.

Resolving by scope

During the construction from the AST, the syntactic scope of each element is eventually known. This allows to map scopes to a list of elements defined in this scope. Definitions from higher scopes are not repeated, instead they are calculated on request. As scopes are lists of scope fragments, for any given scope the set of referable nodes is determined by unifying IDs of all prefixes of the given scope, then resolving the IDs to elements. The Rust implementation is given in lst. 0.22 below.

LSP Server

Section **??** introduced the concept of capabilities in the context of the language server protocol. This section describes how NSL uses the linearization described in sec. **??** to implement a comprehensive set of features. NLS implements the most commonly compared capabilities *Code completion*, *Hover Jump to def*, *Find references*, *Workspace symbols* and *Diagnostics*.

Diagnostics and Caching

NLS instructs the LSP client to notify the server once the user opens or modifies a file. Each notification contains the complete source code of the file as well as its location. NLS subsequently parses and type-checks the file using Nickel's libraries. Since Nickel deals with error reporting already, NLS converts any error generated in these processes into Diagnostic items and sends them to the client as server notifications. Nickel errors provide detailed information about location of the issue as well as possible details which NLS can include in the Diagnostic items.

As discussed in sec. **??** and sec. **??** the type-checking yields a `Completed` linearization which implements crucial methods to resolve elements. NLS will cache the linearization for each processed file. This way it can provide its LSP functions while a file is being edited.

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.

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. **??**. 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.

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.

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. **??** to retrieve symbol names for a reference scope using the method described in sec. **??**. The current scope taken as reference is derived from the item at cursor position.

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.

Listing 0.12 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
```

Listing 0.13 Definition of Linearization structure

```rust
pub trait LinearizationState {}

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

Listing 0.14 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 0.15 Type Definition of Completed state

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

Listing 0.16 Definition of a linearization items TermKind

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

    Usage(UsageState),

    Structure,
}

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

Listing 0.17 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;
}
```

Listing 0.18 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 */ }
    }
```

Listing 0.19 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" }!"
```

Listing 0.20 A record in Nickel

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

Listing 0.21 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
  }
}
```

Listing 0.22 Resolution of all items in scope

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