Contents

Chapter 1

Design implementation of NLS

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 1.5 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. 1.6.

1.1   Key Objectives

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

1.1.1   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. **??**), 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.

1.1.2   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,
2. Hover information,
3. Jump to definition,
4. Find references,
5. Workspace symbols,
6. Diagnostics

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

1.1.3   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.

1.1.4   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.

1.2   Design Decisions

Section **??** introduced several considerations with respect to the implementation of language servers. Additionally, in sec. **??** presents examples of different servers

which guided the decisions made while implementing the NLS. Additionally, in sec. **??** presents examples of different servers which guided the decisions made while implementing the NLS.

1.2.1 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 the concept of `traits` (**traits?**) is fundamental, for the following reasons: 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.

1.2.2 File processing

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

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

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

1.2.3   Code Analysis

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

1.3   High-Level Architecture

This section describes The high-level architecture of NLS. The entity diagram depicted in fig. 1.1 shows the main elements at play.

NLS needs to meet the flexibility and generalizability requirements as discussed in sec. 1.1.3. 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. 1.5. 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 **??** lays out a more detailed plan how this will be achieved.
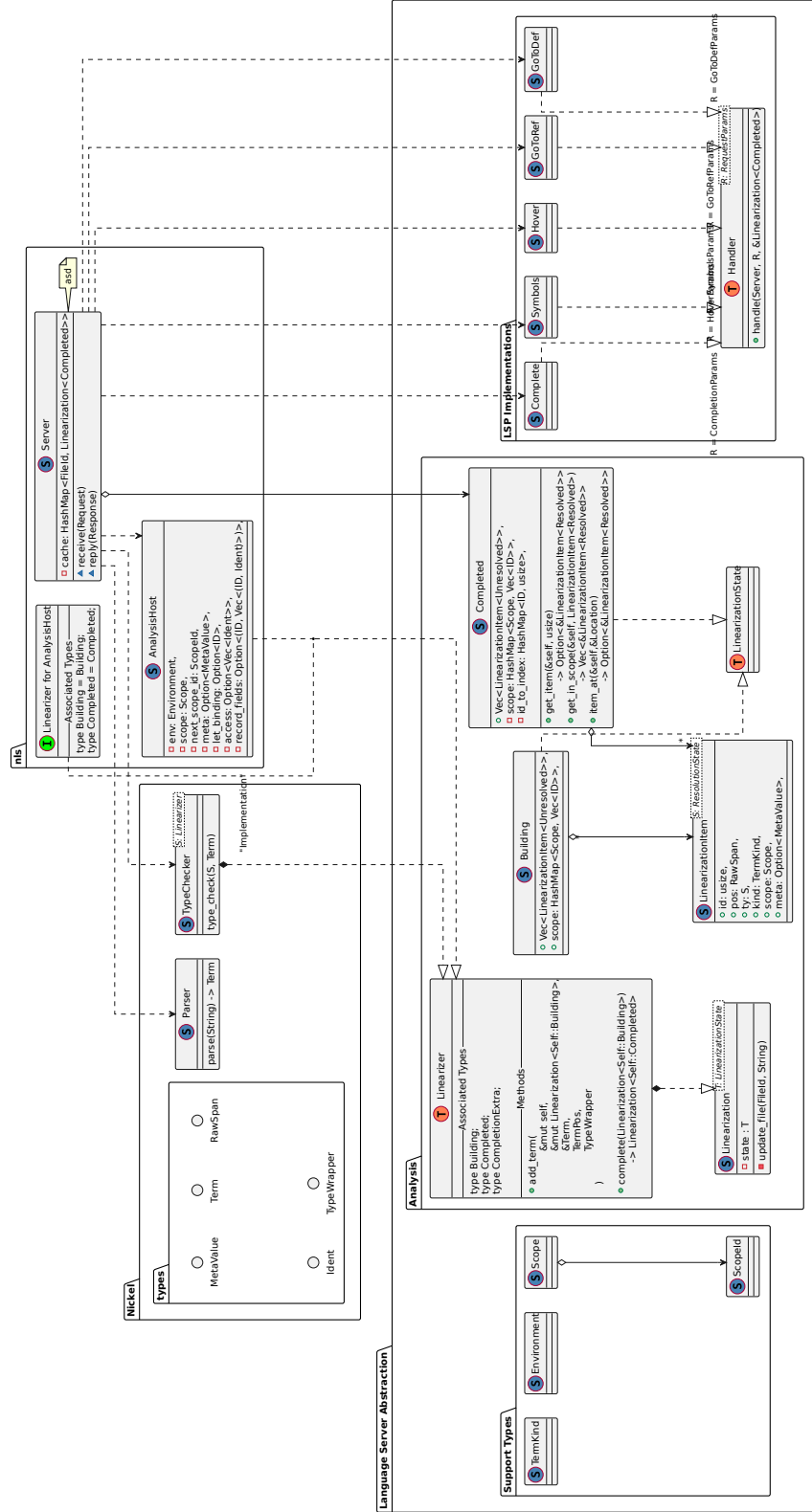
Figure 1.1: Entity Diagram showing the architecture of NLS, explicit dependency arrows omitted for legibility.

1.4 Illustrative example

The example lst. 1.1 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 chema 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 1.1 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
```

1.5   Linearization

The focus of the NLS as presented in this work is to implement a foundational set of LSP features as described in sec. 1.1.2. In order to process these capabilities efficiently as per sec. 1.1.1, NLS needs to store more information than what is originally present in a Nickel AST (cf. sec. **??**), 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. 1.2.3, 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. 1.5.2.

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. 1.5.3). 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. 1.5.4.

1.5.1   States

At its core the linearization in either state is represented by an array of `LinearizationItems` 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.

`LinearizationItems` 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 `LinearizationItems` 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 `ids` 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. 1.2, 1.3, 1.4. Note that only the former is defined as part of the Nickel libraries, the latter are specific implementations for NLS.

Listing 1.2 Definition of Linearization structure

```
pub trait LinearizationState {}

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

Listing 1.3 Type Definition of Building state

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

Listing 1.4 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 {}
```

1.5.2   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.

1.5.2.1   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. 1.5).

Listing 1.5 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. `Usages` 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.

1.5.2.2 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. 1.5.4.3.

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 1.6 shows the scopes for a
simple expression in Nickel explictly.

Listing 1.6 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.

1.5.2.3   Linearizer

The heart of the linearization is the `Linearizer` trait as defined in lst. 1.7. The
`Linearizer` lives in parallel to the `Linearization`. Its methods modify a shared
reference to a `Building Linearization`.

Listing 1.7 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 iden-
    tifier.
    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. 1.5.2.4.3).

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

1.5.2.4 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. 1.8. 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 1.8 Abstract type checking function

```rust
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 1.9 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" }!"
```

1.5.2.4.1 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. 1.9

1.5.2.4.2 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 1.10 A record in Nickel

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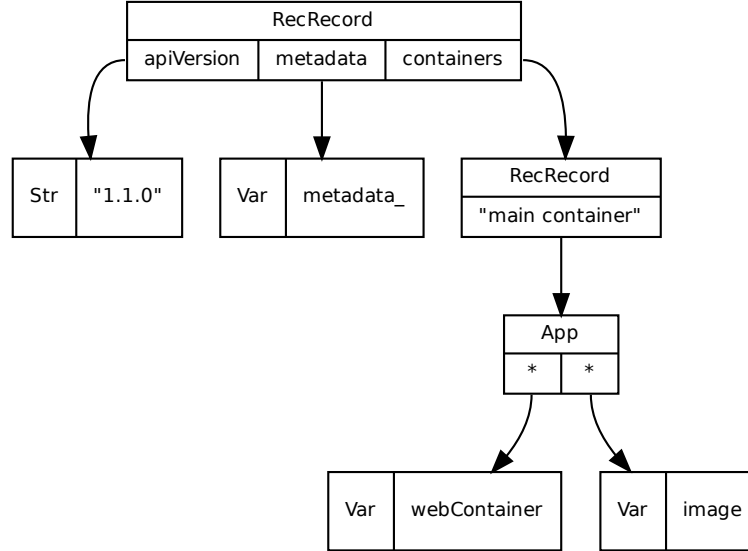

Figure 1.2: AST representation of a record

1.5.2.4.3   Records   Section **??** introduced the AST representation of Records. As suggested by fig. 1.2, 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. 1.3) of the record discussed in lst. 1.10 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. 1.3).

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.

1.5.2.4.4   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.

1.5.2.4.5   Resolution of Record Fields   The AST representation of record destructuring in fig. **??** 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.

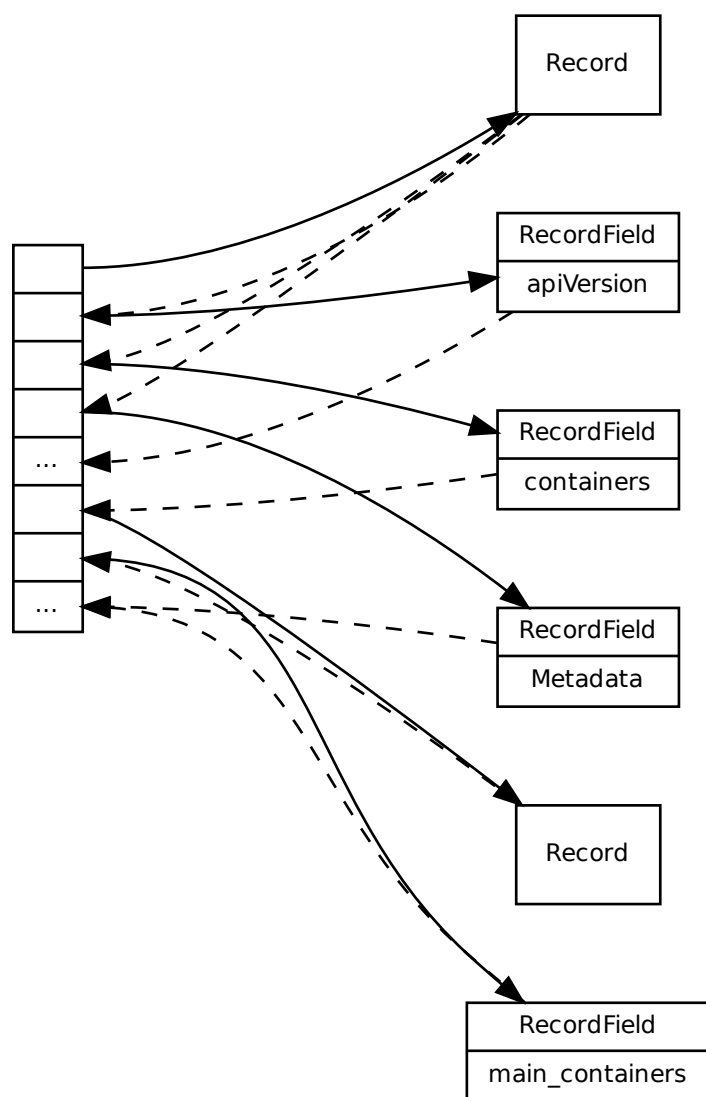Consider the following example lst. 1.11, which is depicted in fig. 1.4

Figure 1.3: Linearization of a record

Listing 1.11 Example of a recursive record

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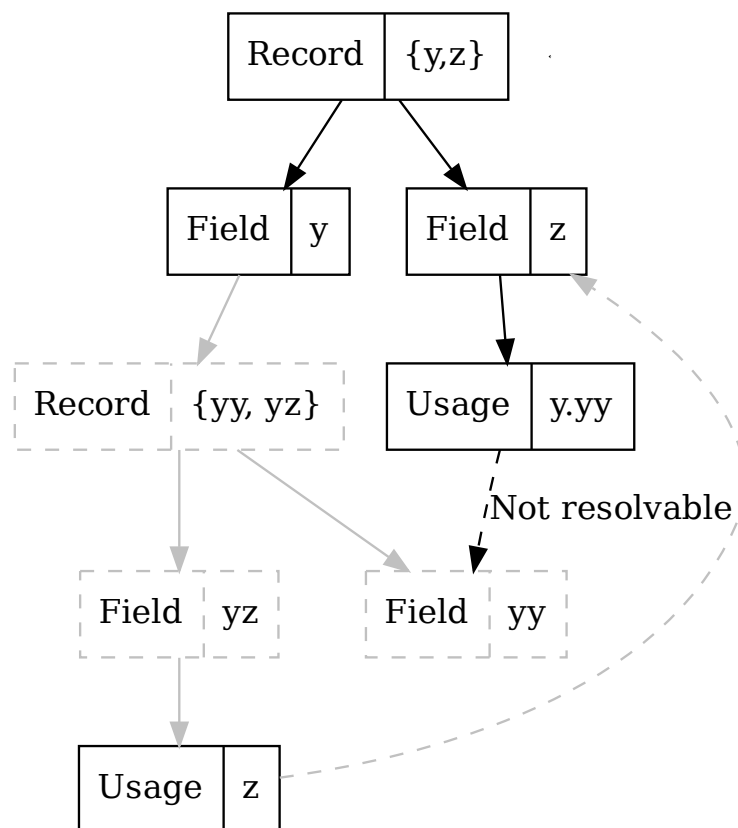

Figure 1.4: Example lock in recursive records. The field 'y.yz' cannot be not be referenced at this point as the 'y' branch has yet to be linearized

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 de-structuring of `y` can not be resolved. While the *field* `y` is known, its value is

not (cf. sec. 1.5.2.4.3), 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. 1.5.3.2.

In fact, NLS linearized all destructor elements as `Deferred` and resolves the correct references later. Figure 1.5 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`.
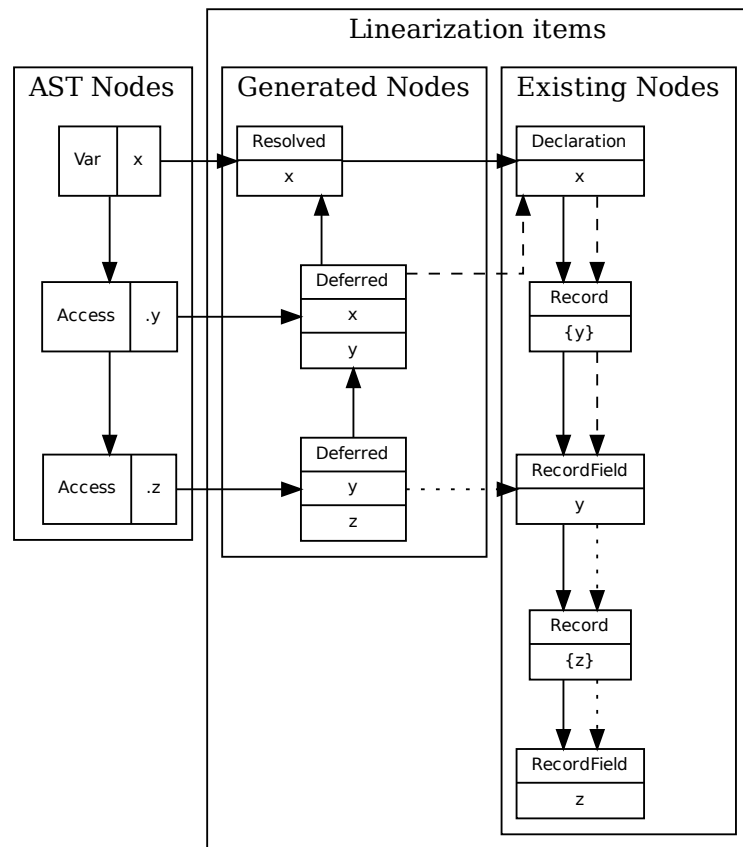

Figure 1.5: Depiction of generated usage nodes for record destructuring

1.5.2.4.6 Metadata In sec. **??** 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`.

1.5.3   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

1.5.3.1   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.

1.5.3.2   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. 1.5, 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`.

1.5.3.3   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.

1.5.4   Resolving Elements

1.5.4.1   Resolving by position

As part of the post-processing step discussed in sec. 1.5.3, the `LinearizationItem`s 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.

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

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. 1.12. 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 1.12 Resolution of item at given position

```rust
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 1.6: Activity diagram of item resolution by position

1.5.4.2   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.

1.5.4.3   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. 1.5.2.2, scopes are lists of scope path elements, where the prefixes correspond to parent scopes. For instance, NLS would represent the example lst. 1.6 as shown in lst. 1.13 below.

Listing 1.13 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 1.13 are `{Declaration("record"), RecordField("key1"), RecordField("key2") }`. The Rust implementation is given in lst. 1.14 below.

Listing 1.14 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()
  }
}
```

1.6 LSP Server Implementation

This section describes how NSL uses the linearization described in sec. 1.5 to implement the set of features proposed in sec. 1.1.2.

1.6.1 Server Interface

As mentioned in sec. 1.2.1 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. 1.1, 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.

1.6.2 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. 1.5 and sec. 1.5.4 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.

1.6.3 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.

1.6.3.1 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. 1.5.4.1. 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.

1.6.3.2   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.

1.6.3.3   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. 1.5.2.2 to retrieve symbol names for a reference scope using the method described in sec. 1.5.4.3. The current scope taken as reference is derived from the item at cursor position.

1.6.3.4   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.