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. 0.1 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. 0.2, 0.3, 0.4. 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. 1. 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.

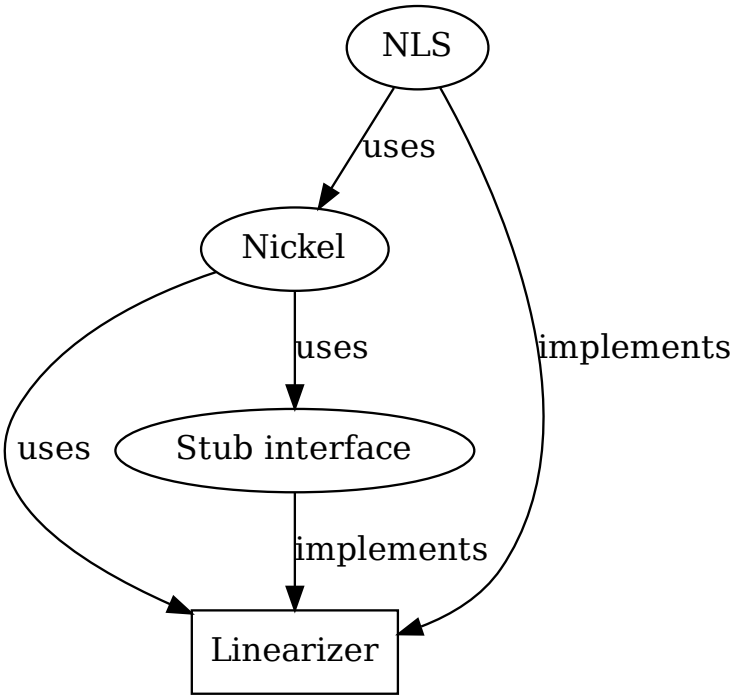

Figure 1: Interaction of Componenets

Usage Graph

At the core the linearization is a simple *linear* structure. Also, in the general case[1] 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.

[1] 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 as **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 in recursive records. It also Integrates a **Record** kind to provide deep 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. Nodes of the AST that do not fit in a usage graph, a wildcard kind `Structure` is applied.

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 can be used to represent this logic. A variable reference always refers to the closest parent node defining the name. 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, scopes are represented by a prefix list. Whenever a new lexical scope is entered the prefix list of the outer scope is extended by a unique identifier. With the example in mind lst. 0.1 contains the defintion of a simple record.

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. 0.5. 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.

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. **??**. 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.

```
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.

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. 0.6 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 Name bindings are equally simple. NLS generates a `Declaration` item for the given identifier and assigns the identifier's position and provided type. Additionally, it associates the identifier with the `id` of the created item in its current environment. If a binding contains a pattern, NLS creates additional items for each matched element. Unfortunately, no types are provided for these by Nickel. Examples of let bindings can be found in use in lst. 0.1

Records

```
{
  apiVersion = "1.1.0",
  metadata = metadata_,
  replicas = 3,
```

```
  containers = {
    "main container" = webContainer image
  }
}
```

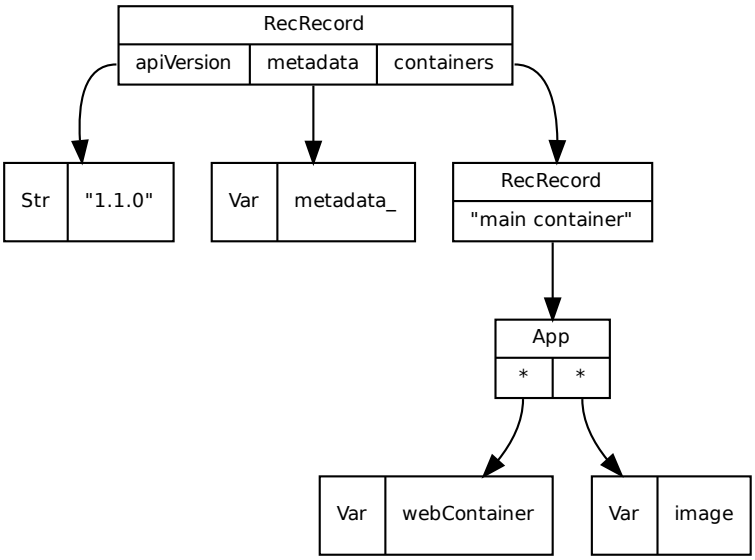

Figure 2: AST representation of a record

Linearizing records proves more difficult. In sec. **??** the AST representation of Records was discussed. As shown by fig. 2, 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 `id`s 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. 3. 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. 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.

Static access

Metadata

Integration with Nickel

Scope

Retyping

Post-Processing

Resolving Elements

Resolving by position

As part of the post-processing step discussed in sec. **??**, 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.

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

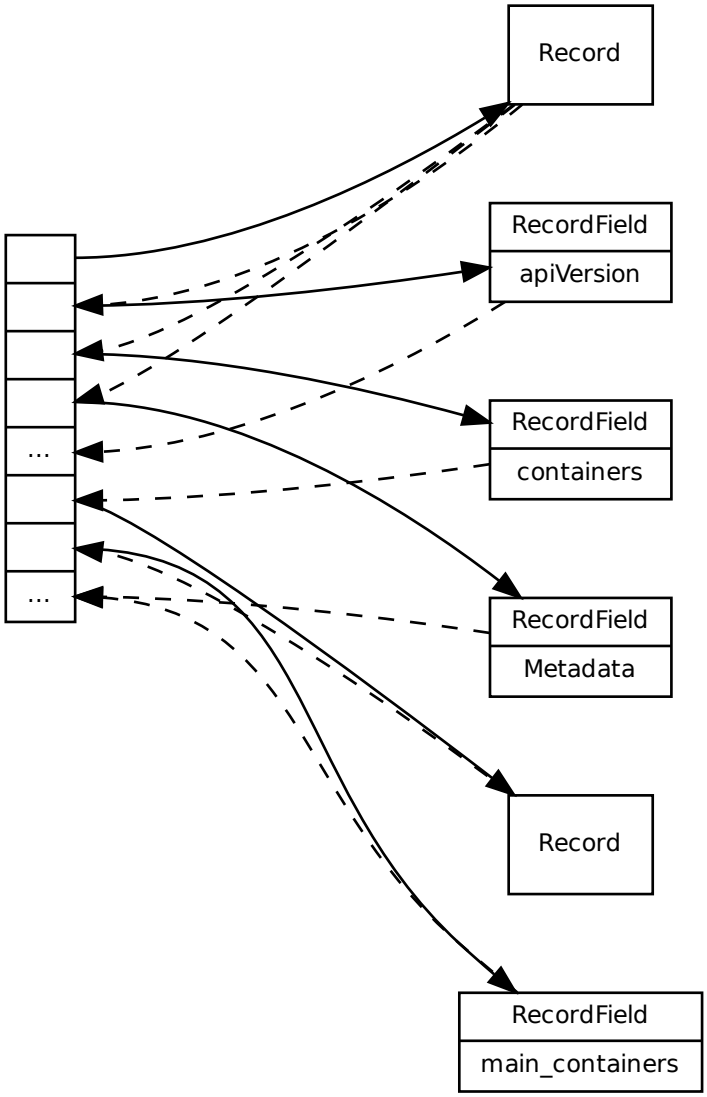

Figure 3: Linearization of a record

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. 0.8. 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. 0.9 below.

LSP Server

Diagnostics and Caching

Capabilities

Hover

Completion

Jump to Definition

Show references

Symbols

Listing 0.1 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

{
  apiVersion = "1.1.0",
  metadata = metadata_,
  replicas = 3,
  containers = {
    "main container" = webContainer  "k8s.gcr.io/#{name_}"
  }
} | #NobernetesConfig
```

Listing 0.2 Definition of Linearization structure

```
pub trait LinearizationState {}

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

Listing 0.3 Type Definition of Building state

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

Listing 0.4 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 0.5 Interface of linearizer trait

```rust
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 0.6 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 0.7 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 0.8 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 0.9 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()
  }
}
```