Chapter 1

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. Section **??** will introduce the main data structure underlying all higher level LSP interactions and how the AST described in sec. **??** is transformed into this form. Finally, the implementation of current LSP features is discussed in sec. **??**.

1.1 Illustrative example

The example lst. 1.1 shows an illustrative high level configuration of a server. Throughout this chapter, different sections about the NSL implementation will refer back to this example.

1.2 Linearization

The focus of the NLS as presented in this work is to implement a working language server with a comprehensive feature set. To answer requests, NLS needs to store more information than what is originally present in a Nickel AST. Apart from missing data, an AST is not optimized for quick random access of nodes based on their position, which is a crucial operation for a language server. To that end NLS introduces an auxiliary data structure, the *linearization*, which is derived from the AST. It represents the original data linearly, performs an enrichment of the AST nodes and provides greater decoupling of the LSP functions from the implemented language. Section **??** details the process of transferring the AST. After NLS parsed a Nickel source files to an AST it starts to fill the linearization, which is in a *building* state during this phase. For reasons detailed in sec. **??**, the linearization needs to be post-processed, yielding a *completed* state. The completed linearization acts as the basis to handle all

supported LSP requests as explained in sec. **??**. Section **??** explains how a completed linearization is accessed.

Advanced LSP implementations sometimes employ so-called incremental parsing, which allows updating only the relevant parts of an AST (and, in case of NLS, the derived linearization) upon small changes in the source. However, an incremental LSP is not trivial to implement. For once, NLS would not be able to leverage existing components from the existing Nickel implementation (most notably, the parser). Parts of the nickel runtime, such as the typechecker, would need to be adapted or even reimplemented to work in an incremental way 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.

1.2.1 States

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

`LinearizationItem`s maintain the position of their AST counterpart, as well as its type. Unlike in the AST, *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, the implementation is based on type-states(**typestate?**). Type-states were chosen over an enumeration bases approach for the additional flexibility they provide to build a generic interface. Thanks to the generic interface, the adaptions to Nickel to integrate NLS are expected to have almost no influence on the runtime performance of the language in an optimized build.

NLS implements separate type-states for the two phases of the linearization: `Building` and `Completed`.

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

post-processing phase: Once fully built, a Building instance is post-processed to get a `Completed` linearization.

Although fundamentally still represented by an array, a completed linearization is optimized for search by positions (in the source file) thanks to sorting and the use of an auxiliary map from `id`s to the new index of items.
Additionally, missing edges in the usage graph have been created and he 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.

1.2.2 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

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.

To accommodate these goals NLS comprises three different parts as shown in fig. 1.1. 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. 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.

1.2.2.1 Usage Graph

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

There are 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.

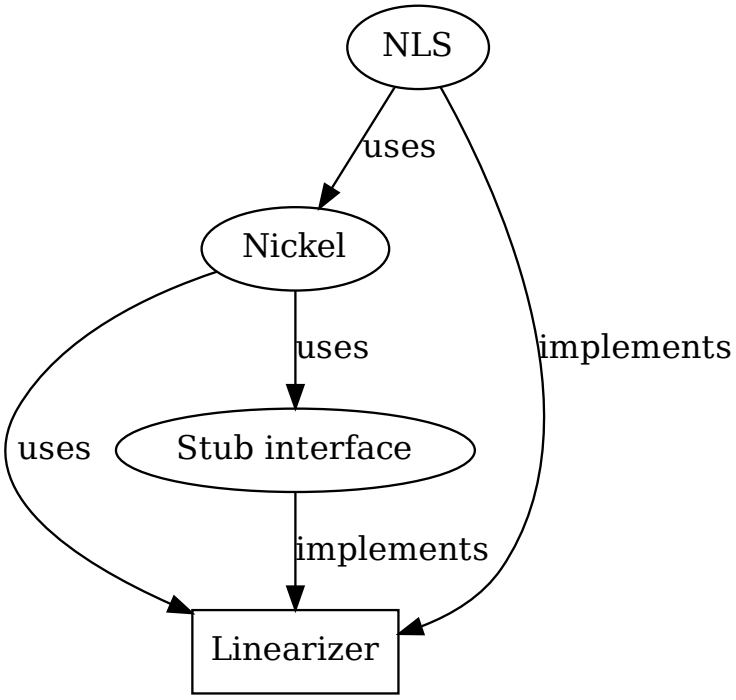

Figure 1.1: Interaction of Componenets

The `TermKind` type is an enumeration which 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 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 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`.

1.2.2.2 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. **??**.

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.

1.2.2.3 Linearizer

The heart of the linearization the `Linearizer` trait as defined in lst. 1.6. 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 of an identifier.
: The reason this method exists is that not all variable definitions have a corresponding AST node but may be part of another node. This is the case with records; Field *names* are not linearized separately but as part of the record. Thus, their type is not known to the linearizer and has to be added explicitly.

`Linearizer::complete` implements the post-processing necessary to turn a final `Building` linearization into a `Completed` one.
: Note that the post-processing might depend on additional data.

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

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

```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>>,
}
```

1.2.2.4 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. 1.7. 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.

1.2.2.4.1 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. 1.8

1.2.2.4.2 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 `Declaration`s.

The same process applies for argument names in function declarations.

1.2.2.4.3 Records Linearizing records proves more difficult. In sec. **??** the AST representation of Records was discussed. As shown 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. For the language server on the

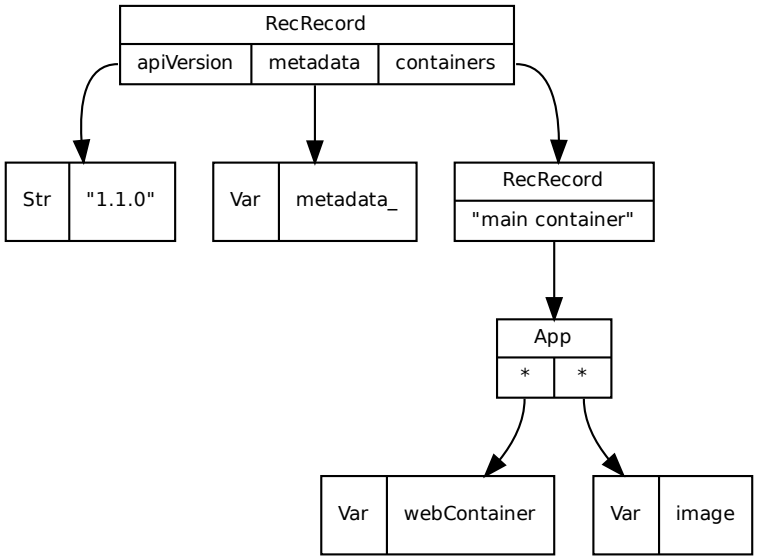

Figure 1.2: AST representation of a record

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. 1.3. 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

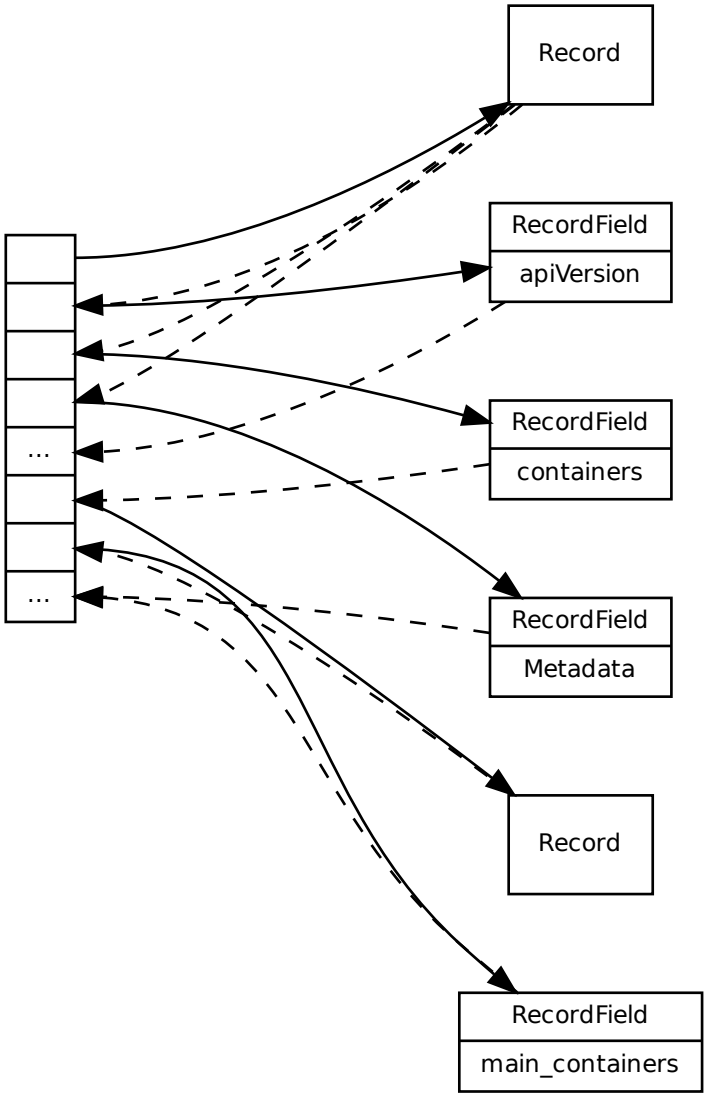

Figure 1.3: Linearization of a record

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.

1.2.2.4.4 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.

Looking at 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 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. A visualization of this is provided in fig. 1.4 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. 1.5 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.

1.2.2.4.5 Metadata In sec. **??** was shown that on the syntax level, metadata "wraps" the annotated value. Conversely, NLS encodes metadata in the

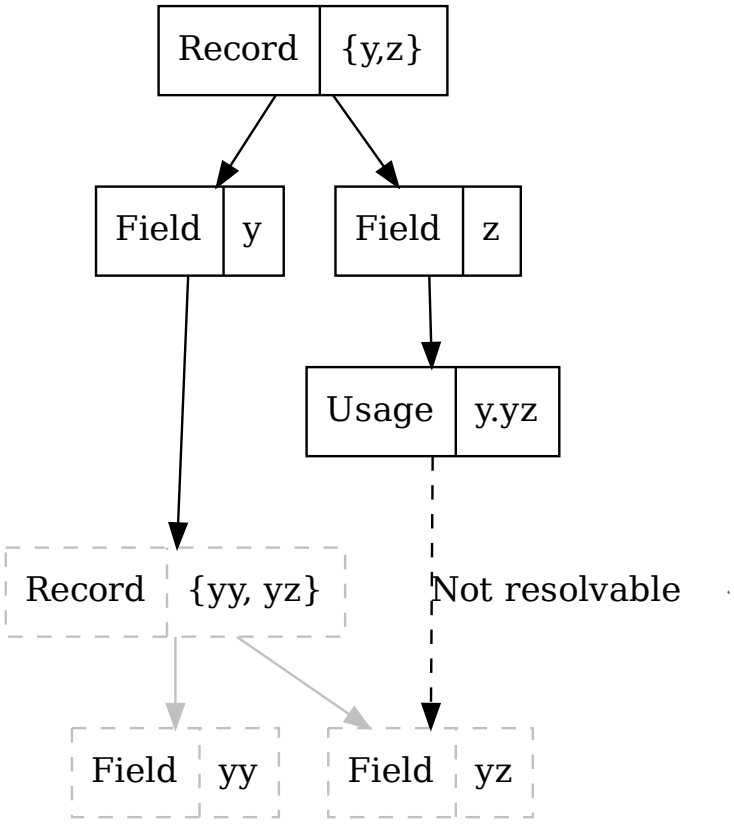

Figure 1.4: Example race condition in recursive records. The field ‘y.yz‘ cannot be not be referenced at this point as the ‘y‘ branch has yet to be linearized

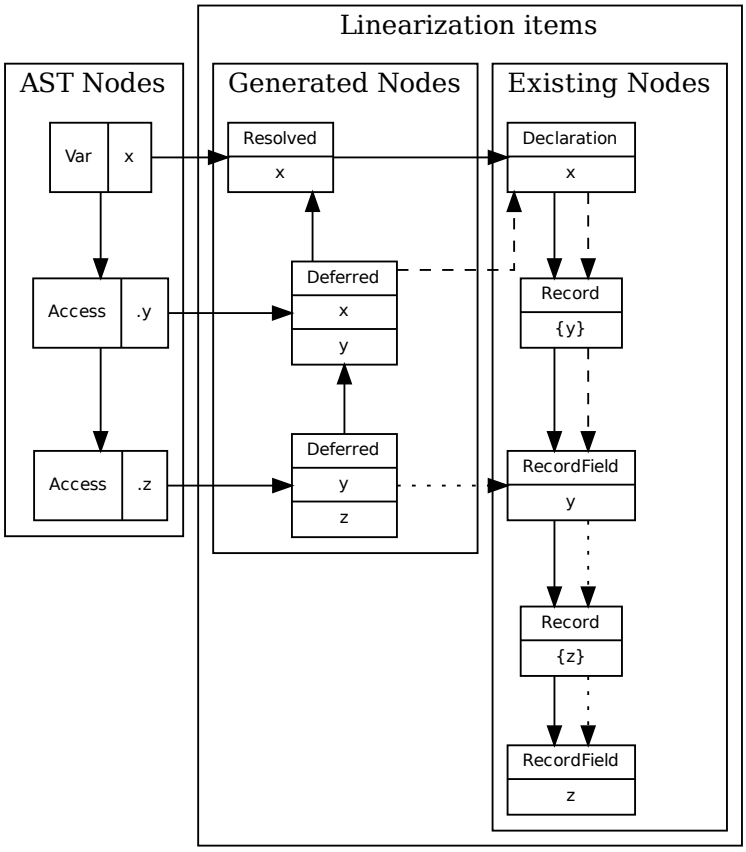

Figure 1.5: Depiction of generated usage nodes for record destructuring

`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.

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`.

1.2.3 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

1.2.3.1 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.

1.2.3.2 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`.

1.2.3.3 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.

1.2.4 Resolving Elements

1.2.4.1 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.

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

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. 1.10. 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`.

1.2.4.2 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.

1.2.4.3 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. 1.11 below.

1.3 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*.

1.3.1 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.

1.3.2 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.

1.3.2.1 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. **??**. 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.

1.3.2.2 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.

1.3.2.3 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. **??** 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.

1.3.2.4 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.

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
```

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 {}
```

Listing 1.5 Definition of a linearization items TermKind

```
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 1.6 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 1.7 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 1.8 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 1.9 A record in Nickel

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

Listing 1.10 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 1.11 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()
  }
}
```