Method

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. Hence, in sec. **??** this chapter will first detail parts of the AST that are of particular interest for the LSP and require special handling. Based on that sec. **??** will introduce the main datastructure underlying all higher level LSP interactions and how the AST is transformed into this form. Finally, in sec. **??** the implementation of current LSP features is discussed on the basis of the previously reviewed compontents.

Nickel AST

Nickel's Syntax tree is a single sum type, i.e. an enumeration of node types. Each enumeration variant may refer to child nodes, representing a branch or hold terminal values in which case it is considered a leaf of the tree. Additionally, nodes are parsed and represented, wrapped in another structure that encodes the span of the node and all its potential children.

Basic Elements

The data types of the Nickel language are closely related to JSON. On the leaf level, Nickel defines `Boolean`, `Number`, `String` and `Null`. In addition to that the language implements native support for `Enum` values. Each of these are terminal leafs in the syntax tree.

Completing JSON compatibility, `List` and `Record` constructs are present as well. Records on a syntax level are HashMaps, uniquely associating an identifier with a sub-node.

These data types constitute a static subset of Nickel which allows writing JSON compatible expressions as shown in lst. 0.1.

Listing 0.1 Example of a static Nickel expression

```
{
  list = [ 1, "string", null],
  "enum value" = `Value
}
```

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

Meta Information

One key feature of Nickel is its gradual typing system [ref again?], which implies that values can be explicitly typed. Complementing type information it is possible to annotate values with contracts and additional meta-data such as documentation, default values and merge priority a special syntax as displayed in lst. 0.2.

Listing 0.2 Example of a static Nickel expression

```
let Contract = {
         foo | Num
             | doc "I am foo",
         hello | Str
               | default = "world"
       }
       | doc "Just an example Contract"
in
let value | #Contract = { foo = 9 }
in value == { foo = 9, hello = "world"}

> true
```

Internally, the addition of annotations wraps the annotated term in a `MetaValue` structure, that is creates an artificial tree node that describes its subtree. Concretely, the expression shown in lst. 0.3 translates to the AST in fig. 1. The green `MetaValue` box is a virtual node generated during parsing and not present in the untyped equivalent.

Listing 0.3 Example of a typed expression

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

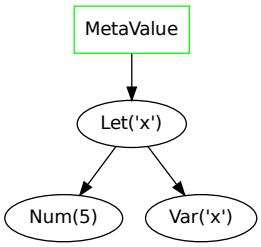

Figure 1: AST of typed expression

Nested Record Access

Nickel supports the referencing of variables which are represented as `Var` nodes that are resolved during runtime. With records bound to a variable, a method to access elements inside that record is required. The access of record members is represented using a special set of AST nodes depending on whether the member name requires an evaluation in which case resolution is deferred to the evaluation pass. While the latter prevents static analysis of any deeper element by the LSP, `StaticAccess` can be used to resolve any intermediate reference.

Notably, Nickel represents static access chains in inverse order as unary operations which in turn puts the terminal `Var` node as a leaf in the tree. Figure 2 shows the representation of the static access perfomed in lst. 0.4 with the rest of the tree omitted.

Listing 0.4 Nickel static access

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

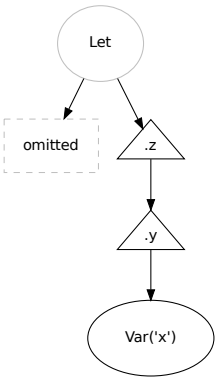

Figure 2: AST of typed expression

Record Shorthand

Nickel supports a shorthand syntax to efficiently define nested records similarly to how nested record fields are accessed. As a comparison the example in lst. 0.5 uses the shorthand syntax which resolves to the semantically equivalent record defined in lst. 0.6

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

Listing 0.5 Nickel record using shorthand

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

Listing 0.6 Nickel record defined explicitly

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

Linearization

Being a domain specific language, the scope of analyzed Nickel files is expected to be small compared to other general purpose languages. NLS therefore takes an *eager approach* to code analysis, resolving all information at once which is then stored in a linear data structure with efficient access to elements. This data structure is referred to 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 an accumulated created incrementally during the linearization process. In particular that is a list of `LinearizationItems` of unresolved type ordered as they appear in a depth-first iteration of the AST. Note that new elements are exclusively appended such that their `id` field during this phase is equal to the elements position at all time. Additionally, the `Building` state maintains the scope structure for every item 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. 0.7, 0.8, 0.9. Note that only the former is defined as part of the Nickel libraries, the latter are specific implementations for NLS.

Listing 0.7 Definition of Linearization structure

```
pub trait LinearizationState {}

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

Listing 0.8 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 0.9 Type Definition of Completed state

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

Transfer from AST

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. Spans may overlap never intersect.

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

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.

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 concrete 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. 0.10 below.

Listing 0.10 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()
  }
}
```

LSP Server

Diagnostics and Caching

Capabilities

Hover

Completion

Jump to Definition

Show references