Contents

Chapter 1

Background

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

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

1.1 Language Server Protocol

Language servers are today's standard of integrating support for programming languages into code editors. Initially developed by Microsoft for the use with their polyglot editor Visual Studio Code[1] before being released to the public in 2016 by Microsoft, RedHat and Codeenvy, the LSP decouples language analysis and provision of IDE-like features from the editor. Developed under open source license on GitHub[2], the protocol allows developers of editors and languages to work independently on the support for new languages. If supported by both server and client, the LSP now supports more than 24 language features[3] including code completion, code navigation facilities, contextual information such as types or documentation, formatting, and more.

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

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

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

1.1.1 JSON-RPC

JSON-RPC (v2) (**json-rpc?**) is a JSON based lightweight transport independent remote procedure call protocol used by the LSP to communicate between a language server and a client.

The protocol specifies the general format of messages exchanges as well as different kinds of messages. The following snippet lst. 1.1 shows the schema for request messages.

Listing 1.1 JSON-RPC Request

```
// Requests
{
  "jsonrpc": "2.0"
, "method": String
, "params": List | Object
, "id": Number | String | Null
}
```

The main distinction in JSON-RPC are *Requests* and *Notifications*. Messages with an `id` field present are considered *requests*. Servers have to respond to requests with a message referencing the same `id` as well as a result, i.e. data or error. If the client does not require a response, it can omit the `id` field sending a *notification*, which servers cannot respond to, with the effect that clients cannot know the effect nor the reception of the message.

Responses, as shown in lst. 1.2, have to be sent by servers answering to any request. Any result or error of an operation is explicitly encoded in the response. Errors are represented as objects specifying the error kind using an error `code` and provide a human-readable descriptive `message` as well as optionally any procedure defined `data`.

Listing 1.2 JSON-RPC Response and Error

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

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

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

1.1.2 Commands and Notifications

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

1.1.2.1 File Notification

1.1.2.1.1 Diagnostics

1.1.2.2 Hover

1.1.2.3 Completion

1.1.2.4 Go-To-*

1.1.2.5 Symbols

1.1.2.6 code lenses

1.1.3 Shortcomings

1.2 Configuration programming languages

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

Static configuration languages such as XML(**xml?**), JSON(**json?**), or YAML(**yaml?**) are language specifications defining how to textually represent structural data used to configure parameters of a system[4]. Applications of configuration languages are ubiquitous especially in the vicinity of software development. While XML and JSON are often used by package managers (**composer?**), YAML is a popular choice for complex configurations such as CI/CD pipelines (**gitlab-runner?**) or machine configurations in software defined networks such as Kubernetes and docker compose.

Such static formats are used due to some significant advantages compared to other formats. Most strikingly, the textual representation allows inspection of a configuration without the need of a separate tool but a text editor and be version controlled using VCS software like Git. For software configuration this is well understood as being preferable over databases or other binary formats. Linux service configurations (files in `/etc`) and MacOS `*.plist` files which can be serialized as XML or a JSON-like format, especially exemplify that claim.

Yet, despite these formats being simple to parse and widely supported (**json?**), their static nature rules out any dynamic content such as generated fields, functions and the possibility to factorize and reuse. Moreover, content validation has to be developed separately, which led to the design of complementary schema specification languages like json-schema (**json-schema?**) or XSD (**xsd?**).

These qualities require an evaluated language. In fact, some applications make heavy use of config files written in the native programming language which gives

[4]some of the named languages may have been designed as a data interchange format which is absolutely compatible with also acting as a configuration language

them access to language features and existing analysis tools. Examples include JavaScript frameworks such as webpack (**webpack?**) or Vue (**vue?**) and python package management using `setuptools`(**setuptools?**).

Despite this, not all languages serve as a configuration language, e.g. compiled languages and some domains require language agnostic formats. For particularly complex products, both language independence and advanced features are desirable. Alternatively to generating configurations using high level languages, this demand is addressed by more domain specific languages. Dhall (**dhall?**), Cue (**cue?**) or jsonnet (**jsonnet?**) are such domain specific languages (DSL), that offer varying support for string interpolation, (strict) typing, functions and validation.

1.2.1 Infrastructure as Code

A prime example for the application of configuration languages are IaaS[5] products. These solutions offer great flexibility with regard to resource provision (computing, storage, load balancing, etc.), network setup and scaling of (virtual) servers. Although the primary interaction with those systems is imperative, maintaining entire applications' or company's environments manually comes with obvious drawbacks.

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

The concept of "Infrastructure as Code" (*IaC*) serves the DevOps principles. IaC tools help to overcome the need for dedicated teams for *Dev*elopment and *Op*erations by allowing to declaratively specify the dependencies, topology and virtual resources. Optimally, different environments for testing, staging and production can be derived from a common base and changes to configurations are atomic. As an additional benefit, configuration code is subject to common software engineering tooling; It can be statically analyzed, refactored and version controlled to ensure reproducibility.

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

To get an idea of how this would look like, lst. 1.3 shows the configuration for a deployment of the Git based wiki server Gollum(**gollum?**) behind a nginx reverse proxy on the AWS network. Although this example targets AWS, Nix itself is platform-agnostic and NixOps supports different backends through various plugins. Configurations like this are abstractions over many manual steps and the Nix language employed in this example allows for even higher level turing-complete interaction with configurations.

[5] Infrastructure as a Service

Listing 1.3 Example NixOps deployment to AWS

```
{
  network.description = "Gollum server and reverse proxy";
  defaults =
    { config, pkgs, ... }:
    {
      deployment.targetEnv = "ec2";
      deployment.ec2.accessKeyId = "AKIA...";
      deployment.ec2.keyPair = "...";
      deployment.ec2.privateKey = "...";
      deployment.ec2.securityGroups = pkgs.lib.mkDefault [ "default" ];
      deployment.ec2.region = pkgs.lib.mkDefault "eu-west-1";
      deployment.ec2.instanceType = pkgs.lib.mkDefault "t2.large";
    };

  gollum =
    { config, pkgs, ... }:
    {
      services.gollum = {
        enable = true;
        port = 40273;
      };
      networking.firewall.allowedTCPPorts = [ config.services.gollum.port ];
    };

  reverseproxy =
    { config, pkgs, nodes, ... }:
    let
      gollumPort = nodes.gollum.config.services.gollum.port;
    in
    {
      deployment.ec2.instanceType = "t1.medium";
      services.nginx = {
        enable = true;
        virtualHosts."wiki.example.net".locations."/" = {
          proxyPass = "http://gollum:${toString gollumPort}";
        };
      };
      networking.firewall.allowedTCPPorts = [ 80 ];
    };
}
```

Similarly, tools like Terraform(**terraform?**), or Chef(**chef?**) use their own DSLs and integrate with most major cloud providers. The popularity of these products[6], beyond all, highlights the importance of expressive configuration formats and their industry value.

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

Finally, descriptive data formats for cloud configurations allow mitigating security risks through static analysis. Yet, as recently as spring 2020 and still more than a year later dossiers of Palo Alto Networks' security department Unit 42 (**pa2020H1?**) show that a majority of public projects uses insecure configurations. This suggests that techniques(**aws-cloud-formation-security-tests?**) to automatically check templates are not actively employed, and points out the importance of evaluated configuration languages which can implement passive approaches to security analysis.

1.2.2 Nickel

1.2.2.1 Gradual typing

1.2.2.1.1 Row types

1.2.2.2 Contracts

In addition to a static type-system Nickel integrates a contract system akin what is described in (**cant-be-blamed?**). First introduced by Findler and Felleisen, contracts allow the creation of runtime-checked subtypes. Unlike types, contracts check an annotated value using arbitrary functions that either pass or *blame* the input. Contracts act like assertions that are automatically checked when a value is used or passed to annotated functions.

For instance, a contract could be used to define TCP port numbers, like shown in lst. 1.4.

Listing 1.4 Sample Contract ensuring that a value is a valid TCP port number

```
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 8080 | #Port
```

Going along gradual typing, contracts pose a convenient alternative to the `newtype` pattern. Instead of requiring values to be wrapped or converted into custom types, contracts are self-contained. As a further advantage, multiple contracts can be applied to the same value as well as integrated into other higher level contracts. An example can be observed in lst. 1.5

Listing 1.5 More advaced use of contracts restricting values to an even smaller domain

```
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 UnprivilegedPort = contracts.from_predicate (
  fun value =>
    (value | #Port) >= 1024
  )
in
let Even = fun label value =>
  if value % 2 == 0 then value
  else
    let msg = "not an even value" in
    contracts.blame_with msg label
in

8001 | #UnprivilegedPort
     | #Even
```

Notice how contracts also enable detailed error messages (see lst. 1.6) using custom blame messages. Nickel is able to point to the exact value violating a contract as well as the contract in question.

Listing 1.6 Example error message for failed contract

```
error: Blame error: contract broken by a value [not an even value].
    - :1:1
   |
 1 | #Even
   | ----- expected type
   |
    - repl-input-34:22:1
   |
22 | - 8001 | #UnprivilegedPort
   |   ---- evaluated to this expression
23 | |      | #Even
   | -------------^ applied to this expression

note:
    - repl-input-34:23:8
   |
23 |      | #Even
   |        ^^^^^ bound here
```

1.2.2.3 Nickel AST

Nickel's syntax tree is a single sum type, i.e., an enumeration of node types. Each enumeration variant may refer to child nodes, representing a branch or hold terminal values in which case it is considered a leaf of the tree. Additionally, tree nodes hold information about their position in the underlying code.

1.2.2.3.1 Basic Elements The primitive values of the Nickel language are closely related to JSON. On the leaf level, Nickel defines `Boolean`, `Number`, `String` and `Null`. In addition to that the language implements native support for `Enum` values which are serialized as plain strings. Each of these are terminal leafs in the syntax tree.

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

These data types constitute a static subset of Nickel which allows writing JSON compatible expressions as shown in lst. 1.7.

Listing 1.7 Example of a static Nickel expression

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

Building on that Nickel also supports variables and functions.

1.2.2.3.2 Identifiers The inclusion of Variables to the language, implies some sort of identifiers. Such name bindings can be declared in multiple ways, e.g. `let` bindings, function arguments and records. The usage of a name is always parsed as a single `Var` node wrapping the identifier. Span information of identifiers is preserved by the parser and encoded in the `Ident` type.

Listing 1.8 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>
```

1.2.2.3.3 Variable Reference Let bindings in their simplest form merely bind a name to a value expression and expose the name to the inner expression. Hence, the `Let` node contains the binding and links to both implementation and scope subtrees. The binding can be a simple name, a pattern or both by naming the pattern as shown in lst. 1.8.

Listing 1.9 Parsed representation of functions with multiple arguments

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

Functions in Nickel are curried lambda expressions. A function with multiple arguments gets broken down into nested single argument functions as seen in lst. 1.9. Function argument name binding therefore looks the same as in `let` bindings.

1.2.2.3.4 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 metadata such as contracts, documentation, default values and merge priority using a special syntax as displayed in lst. 1.10.

Listing 1.10 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`, an additional tree node which describes its subtree. The expression shown in lst. 1.11 translates to the AST in fig. 1.1.

Listing 1.11 Example of a typed expression

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

1.2.2.3.5 Nested Record Access Nickel supports both static and dynamic access to record fields. If the field name is statically known, the access is said to be *static* accordingly. Conversely, if the name requires evaluating a string from an expression the access is called *dynamic*. An example is given in lst. 1.12

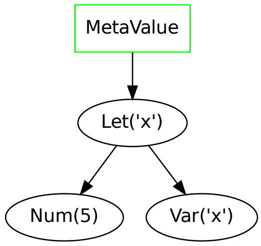

Figure 1.1: AST of typed expression

Listing 1.12 Examples for static and dynamic record access

```
let r = { foo = 1, "bar space" = 2} in
r.foo // static
r."bar space" // static
let field = "fo" ++ "o" in r."#{field}" // dynamic
```

The destruction of record fields is represented using a special set of AST nodes depending on whether the access is static or dynamic. Static analysis does not evaluate dynamic fields and thus prevents the analysis of any deeper element starting with dynamic access. Static access however 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 1.2 shows the representation of the static access performed in lst. 1.13 with the rest of the tree omitted.

Listing 1.13 Nickel static access

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

1.2.2.3.6 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. 1.14 uses the shorthand syntax which resolves to the semantically equivalent record defined in lst. 1.15

Listing 1.14 Nickel record defined using shorthand

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

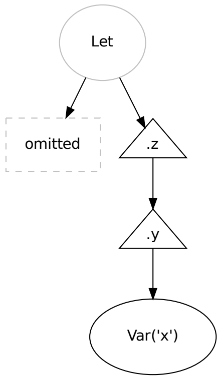

Figure 1.2: AST of typed expression

Listing 1.15 Nickel record defined explicitly

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

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