Contents

Chapter 1

Evaluation

Section **??** described the implementation of the Nickel Language Server addressing the first research question stated in sec. **??**. Proving the viability of the result and answering the second research question demands an evaluation of different factors.

Earlier, the most important metrics of interest were identified as:

Usability What is the real-world value of the language server?
Does it improve the experience of developers using Nickel? NLS offers several features, that are intended to help developers using the language. The evaluation should assess whether developers experience any help due to the use of the server.
Does NLS meet its users' expectations in terms of completeness and correctness and behavior? Being marketed as a Language Server, invokes certain expectations due to previous experience with other languages and language servers. Here, the evaluation should show whether NLS lives up to the expectations of its users.

Performance What are the typical latencies of standard tasks? In this context *latency* refers to the time it takes from issuing an LSP command to return of the reply by the server. The JSON-RPC protocol used by the LSP is synchronous, i.e. requires the server to return results of commands in the order it received them. Since most commands are sent implicitly, a quick processing is imperative to avoid commands queuing up.
Can single performance bottlenecks be identified? Single commands with excessive runtimes can slow down the entire communication resulting in bad user experience. Identified issues can guide the future work on the server.
How does the performance of NLS scale for bigger projects? With increasing project sizes the work required to process files increases as well. The evaluation should allow estimates of the sustained performance in real-world scenarios.

Answering the questions above, this chapter consists of two main sections. The first section sec. 1.1 introduces methods employed for the evaluation. In particular, it details the survey (sec. 1.1.1) which was conducted with the intent

to gain qualitative opinions by users, as well as the tracing mechanism (sec. 1.1.2) for factual quantitative insights. Section 1.3 summarises the results of these methods.

1.1 Methods

1.1.1 Qualitative

1.1.2 Quantitative

1.2 Process

1.3 Results

1.3.1 Qualitative

1.3.2 Quantitative

1.4 Discussion

This section discusses the issues raised during the survey and uncovered through the performance tracing. In the first part the individual findings are summarized and if possible grouped by their common cause. The second part addresses each cause and connects it to the relevant architecture decisions, while explaining the reason for it and discussing possible alternatives.

During the qualitative evaluation several features did not meet the expectations of the users. The survey also hinted performance issues that were solidified by the results of the quantitative analysis.

1.4.1 Diagnostics

First, participants criticized sec. 1.4.1 the diagnostics feature for some unhelpful error messages and specifically for not taking into account Nickel's hallmark feature, Contracts sec. **??**. While Contracts are a central element of Nickel and relied upon to validate data, the language server does not actually warn about contract breaches. Yet, while contracts and their application looks similar to types, contracts are a dynamic language element which are dynamically applied during evaluation. Therefore it is not possible to determine whether a value conforms to a contract without evaluation of the contract. NLS's is integrated with Nickel's type-checking mechanism which precedes evaluation and provides only a static representation of the source code. In order to support diagnostics for contracts NLS would need to locally evaluate arbitrary code that makes up contracts. However, contracts can not be evaluated entirely locally as they may transitively depend on other contracts. This is particularly true for a file's output value. Additionally, Contracts can implement any sort of complex computation including unbound recursion. Due to these caveats, evaluating contracts as part of NLS's analysis implies the evaluation of the entire code which was considered a possibly significant impact to the performance. As layed out above evaluating contracts locally is no option either. It is not only challenging to collect the

minimal context of the Contract, the context may in fact be the entire program. An alternative option is to provide the ability to apply contracts manually using an LSP feature called “Code Lenses.” Code Lenses are displayed by editors as annotations allowing the user to manually execute an associated action.

1.4.2 Cross File Navigation

In both cases `Jump-To-Definition` and `Find-References` surveyed users requested support for cross file navigation. In particular, finding the definition of a record field of an imported record should navigate the editor to the respective file as symbolized in lst. 1.1.

Listing 1.1 Minimal example of cross file referencing

```
// file_a.ncl

let b = import "./b.ncl" in b.field
                              |
                              +------+
                                     |
-----------------------------------  |
                                     |
// file_b.ncl                        |
                                     |
{                                    |
  field = "field value";             |
}  ^                                 |
   +---------------------------------+
```

The resolution of imported values is done at evaluation time, the AST therefore only contains nodes representing the concept of an import but no not reference elements of that file. NLS does ingest the the AST without resolving these imports manually. The type checking module underlying NLS still recurses into imported files to check their formal correctness. As a result it would be possible for a NLS to resolve these links as an additional step in the post processing by either inserting atificial linearization items sec. **??** or merging both files linearization entirely.

1.4.3 Autocompletion

Another criticized element of NLS was the autocompletion feature. In the survey, participants mentioned the lack of additional information and distinction of elements as well as NLS inability to provide completion for record fields. In Nickel, record access is declared by a period. An LSP client can to configured to ask for completions when such an access character is entered additionally to manual requests by the user. The language server is then responsible to provide a list of completion candidates depending on the context, i.e. the position. [Section #sec:completion] describes how NLS resolves this kind of request. NLS just lists all identifiers of declarations that are in scope at the given position. Notably, it does not take the preceding element into account as additional context. To

support completing records, the server must first be aware of separating tokens such as the period symbol, check whether the current position is part of a token that is preceded by a separator and finally resolve the parent element to a record.

1.4.4 Performance

In the experience survey performance was pointed out as a potential issue. Especially in connection with the diagnostics and hover feature. NLS was described to "queue a lot of work and not respond" and show different performance signatures depending on its usage. While comands resolved "instantaniously" on unmodified files, editing a file causes high CPU usage and generally "very slow" responses. An analysis of the measured runtime of 16761 requests confirmed that observation. Both Hover and Update requests showed a wide range of latencies with some reaching more than two minutes. However, the data distribution also confirmed that latencies for most requests except `didOpen` are distributed well below one millisecond. The `didOpen` requests which are associated with the linearization process sec. **??** peak around $1ms$ but longer latencies remain frequent fig. **??**. Lookig deeper into the individual features, reveals signs of the aforementioned "stacking." As discussed in sec. **??** subsequent requests exhibit increasing processing times especially during peak usage.

This behavior is caused by the architecture of the LSP and NLS' processing method. The Language Server Protocol is a synchronous protocol which requires the processing of all requests FIFO order. In effect, every request is delayed until previous requests are handled. This effect is particularly strong as the server is faced with a high volume. In the case of the trace for `didOpen` events the delay effect is greater than for other methods as `didOpen` is associated with a full analysis of the entire file. NLS architecture is heaviliy influenced by the desire to reuse as many elements of the Nickel runtime as possible to maintain feature parity with the evolving language core. Consequently file updates invoke a complete eager analysis of the contents; The entire document is parsed, typechecked and recorded to a linearization everytime. In contrast, all other methods rely on the linearization of a document which allows them to use a binary search to efficiently lookup elements in logarithmic time. Additionally, all requests regardless of their type are subject to the same queue. Given that `didOpen` requests make up > 80 of the recorded events, suggests that other events are heavily slowed down colaterally.

Multiple ways exists to address this issue by reducing the average queue size. The most approachable way to reduce queue sizes is to reduce the number of requests the server needs to handle. The `didOpen` trace elements actually represents the joint processing path of initial file openings, and changes. NLS configures clients to signal changes both on save as well as following editor defined "change." The fact that it is the editor's responsibility to define what constitutes a changes means that some editors send invoke the server on every key press. In fig. **??** signs for such a behavior can be seen as local increases of processing time as the document grows. Hence, restricting analysis to happen only as the user saves the document could potentially reduce the load of requests substatially. Yet, many users prefered automatic processing to happen while they type. To server this pattern, NLS could implement a debouncing mechanism for the processing of document changes. The messages associated to document changes and openings

are technically no requests but notifications. The specification of JSON-RPC which the LSP is based on defines that notifications are not allowing a server response. Clients can not rely on the execution of associated procedures. In effect, a language server like NLS, where each change notification contains the entire latest document, may skip the processing of changes. In practice, NLS could skip such queue items if a more recent version of the file is notified later in the queue. The queue size can also be influenced by reducing the processing time. Other language servers such as the rust-analyzer (**rust-analyzer?**) chose to process documents lazily. Update requests incrementally change an internal model which other requests use as a basis to invoke targeted analysis, resolve elements and more. The entire model is based on in incremental computation model which automates memoization of requests. This method however requires rust-analyzer to reimplement core components of rust to support incrementality. Therefore if one accepts to implement an incremental model of Nickel (including parsing and type checking) to enable incremental analysis in NLS, switching to a lazy model is a viable method to reduce the processing time of change notifications and shorten the queue.