Chapter 1

Related work

The Nickel Language Server follows a history of previous research and development in the domain of modern language tooling and editor integration. Most importantly, it is part of a growing set of LSP integrations. As such, it is important to get a picture of the field of current LSP projects. This chapter will survey a varied range of popular language servers, compare common capabilities, and implementation approaches. Additionally, this part aims to recognize alternative approaches to the LSP, in the form of legacy protocols, extensible development platforms LSP extensions and the emerging Language Server Index Format.

1.1 Previous Approaches

1.1.1 IDEs

Before the invention of the Language Server Protocol, language intelligence used to be provided by an IDE. Yet, the range of officially supported languages remained relatively small (**intellij-supported-languages?**). While integration for popular languages was common, top-tier support for less popular ones was all but guaranteed and relied mainly on community efforts. In fact Eclipse(**eclipse-a-platform?**,eclipse-www), IntelliJ(**intelliJ?**), and Visual Studio(**VisualStudio?**), to this day the most popular IDE choices, focus on a narrow subset of languages, historically Java and .NET. Additional languages can be integrated by custom (third-party) plugins or derivations of the base platform ((**jetbrains-all-products?**)). Due to the proprietary nature of some of these products, plugins are not compatible between different platforms. Many less popular languages therefore saw redundant implementations of what is essentially the same. For Haskell separate efforts produced an eclipse based IDE (**haskell-ide-eclips?**), as well as independent IntelliJ plugins (**HaskForce?**). Importantly, the implementers of the former reported troubles with the language barrier between Haskell and the Eclipse base written in Java.

The Haskell language is an exceptional example since there is also a native Haskell IDE(**haskell-for-mac?**) albeit that it is available only to the MacOS operating system. This showcases the difficulties of language tooling and its provision.

In general, developing language integrations, both as the vendor of an IDE or a third-party plugin developer requires extensive resources. Table 1.1 gives an idea of the efforts required. Since the IntelliJ platform is based on the JVM, its plugin system requires the use of JVM languages (**custom-language-support?**), making it hard to reuse the code of e.g. a reference compiler or interpreter. The Rust and Haskell integrations for instance contain at best only a fraction of code in their respective language.

Table 1.1: Comparison of the size for different IntelliJ platform plugins

Plugin	lines of code
intellij-haskell	17249 (Java) + 13476 (Scala) **+ 0 (Haskell)**
intellij-rust	229131 (Kotlin) + 3958 (Rust)
intellij-scala	39382 (Java) + 478904 (Scala)
intellij-kotlin	182372 (Java) + 563394 (Kotlin)
intellij-community/python	47720 (C) + 248177 (Java) + 37101 (Kotlin) + 277125 (Python)

Naturally, development efforts at this size tend to gravitate around the most promising solution, stifling the progress on competing platforms (**intellij-comparison-eclipse?**). Editor-specific approaches also tend to lock-in programmers into a specific platform for its language support regardless of their personal preference.

1.1.2 IDE Abstraction

1.1.2.1 Monto

The authors of the Monto project(**monto-disintegrated?**) call this the "IDE Portability Problem." They compare the situation with the task of compiling different high level languages to a set of CPU architectures. In Compilers, the answer to that problem was the use of an intermediate representation (IR). A major compiler toolchain making use of this is the LLVM (**llvm?**). Compiler frontends for different languages – e.g. Clang(**clang?**), Rustc(**rustc?**), NVCC(**nvcc?**),... – compile input languages into LLVM IR, a low level language with additional capabilities to provide optimization hints but independent of a particular architecture. LLVM performs optimization of the IR and passes it to a compiler backend which in turn generates bytecode for specific architectures e.g. `x86_64`, `MIPS`, `aarch64`, `wasm`, etc. Notably through this mechanism, languages gain support for a range of architectures and profit from existing optimizations developed for the IR.

With Monto, Kreidel et al propose a similar idea for IDE portability. The paper describes the *Monto IR* and how they use a *Message Broker* to receive events from the Editor and dispatch them to *Monto Services*.

The Monto IR is a language-agnostic and editor-independent tree-like model serialized as JSON. Additionally, the IR maintains low level syntax highlighting information (font, color, style, etc.) but leaves the highlighting to the language specific service.

The processing and modification of the source code and IR is performed by *Monto Services*. Services implement specific actions, e.g. parsing, outlining or highlighting. A central broker connects the services with each other and the editor.

Since Monto performs all work on the IR, independent of the editor, and serializes the IR as JSON messages, the language used to implement *Monto Services* can be chosen freely giving even more flexibility.

The Editor extension's responsibility is to act as a source and sink for data. It sends Monto compliant messages to the broker and receives processing results such as (error) reports. The communication is based on the ZeroMQ[zeromq] technology which was chosen because it is lightweight and available in manly languages (**monto-disintegrated?**) allowing to make use of existing language tools.

1.1.2.2 Merlin

The Merlin tool (**merlin-website?**) is in many ways a more specific version of the idea presented in Monto. Merlin is a language server for the Ocaml language, yet predates the Language Server Protocol.

The authors of Merlin postulate that implementing "tooling support traditionally provided by IDEs" for "niche languages" demands to "share the language-awareness logic" between implementations. As an answer to that, they describe the architecture of Merlin in (**merlin?**).

Similarly to Monto, Merlin separates editor extensions from language analysis. However, Merlin uses a command line interface instead of message passing for interaction. Editor extensions expose the server functions to the user by integrating with the editor.

The Merlin server hand provides a single optimized implementation of code intelligence for Ocaml. Since all resources could be put to a single project, multiple iterations of performance improvements were done on Merlin. It now supports partial, incremental parsing and type-checking which allows users to query information even about incomplete or incorrect programs.

Notably, being written in Ocaml, Merlin can make use of existing tools of the Ocaml language. In fact, its parser and type-checker are based on the respective original implementations. The Merlin project did however have to adapt the Ocaml type-checker to support the aforementioned incrementality. Changes are made against a copy of the relevant modules shipped with Merlin which facilitates keeping up with the latest developments of the language.

While Merlin serves as a single implementation used by all clients, unlike Monto it does not specify a language independent format, or service architecture.

1.2 Language Servers

1.2.1 Considerable dimensions

1.2.1.1 Language Complexity

1.2.1.2 LSP compliance

1.2.1.3 Features

1.2.1.4 File processing

1.2.1.4.1 Incremental

1.2.1.4.2 Full

1.2.2 Comparative Projects

1.2.3 Honorable mentions

1.3 Alternative approaches

1.3.1 Platform plugins

1.3.2 Legacy protocols

1.3.3 LSP Extensions

1.3.4 LSIF