Modular Language Server: Design and Implementation

Innopolis University

Thesis submitted to The Innopolis University in conformity with the requirements for the degree of Bachelor of Science.

presented by

Mike Lubinets

supervised by

Eugene Zouev

12/20/2017

Modular Language Server: Design and Implementation

Contents

Abstract

Nowadays we have a lot of mature toolkits and integrated development environments for a set of widely used programming languages evolved through years to fulfill needs of the most of software development industry.

However, such software products are highly complex and often have a monolithic architecture, usually standalone from the language compiler infrastructure which makes it hard to replace key components or implement support for additional languages.

In this paper we will see how conventional compilers have been architectured and what complications this consequently brought to modern IDE implementations, review the modern approach to compiler construction and implement a flexible distributed integrated development environment with the
Language Server Protocol, for a multi paradigm SLang[1] programming language.

Chapter 1

Literature Review

1.1 Conventional compilers in the modern world

Since the middle of 20^{th} century researchers and the industry have done a great work in compilers development focusing on the main goal of classic compilation problem: producing fast and effective object code for execution on a virtual machine or a microprocessor.

However, the other compilers‘ capability as developer‘s code inspection instruments able of providing comprehensive information of code semantics remains uncommon and rarely well-developed in the modern compilers of popular programming languages.

The most common illustration of this problem can be found in an average Java developer‘s set of instruments:

Java code is usually compiled with Sun Java Compiler. “Being a monolithic program, constructed as a ‘black box‘”[2], Sun Java Compiler can only accept the input code and produce optimized JVM byte code.

Yet a modern development environment includes a set of tools for programming assistance and requires advanced language syntax and semantics inspection, which is not possible without building a Semantic Representation[2]. To build it one needs to re-implement core functionality of a compiler.

It‘s easy to understand the very reason of the issue looking back to the past: traditionally programs have been considered as mere text objects to be converted into an executable code. According to this assumption, compilers were designed in a very logical way: they haven‘t maintained any semantic representation of source code, only some low-level internal IR.

These compilers‘ IRs have very limited set of use cases[2, 3], moreover they are good for the only task: object code generation for several microprocessor architectures. Also compiler‘s IR is not stable and tends to change very actively

during compiler development[4]. Hence the internal compiler's IR can't really be used to build good and reliable development tools.

The situation is even more frustrating with C++ tooling: the language syntax and semantics is a lot more complicated than Java's, so building a custom compiler is a very complex task.

As a result, there is a notable lack of instruments for C++[3], and existing ones are pretty sophisticated: JetBrains CLion IDE implements its own parser and semantic analyzer to build auto-completion, refactoring and static analysis tools upon their own C++ SR. Being a complex software product, CLion's parser tends to have its own misfeatures and a few month implementational lag to fully adapt for new standards.

Microsoft Visual Studio "suffers" from this too: VC++ generates IR that is useful only for code generation: it is fragmented and very low-level. The C&C++ IntelliSense tooling in Microsoft Visual Studio IDE is implemented as a solution separate from VC++ compiler.

1.2 Modern compilers and SR: a new hope

In spite of the fact that traditional compilers are widely used today, their lack of IDE integration capabilities were realized long ago and currently a lot of new languages are aiming to implement a Semantic Representation as a stable IR shared with external tools.

Following [2, 3], unlike a traditional compiler IR, Semantic Representation contains a full knowledge of a program, including the aspects that are implicit in the source code. This trait enables some pretty powerful opportunities based on semantics analysis:

- Code generation
- Distributed (or recursive) Validation
- Human understandable visualization
- Static analysis
- Program interpretation
- Semantic Search: the very powerful technique of querying code semantic objects ("find all classes derived from class C that do not override the virtual function f")

There are two main ways to represent an SR and share it with SR clients: to provide an access API operating on an SR (proprietary) binary format [4, 5],

relational database representing a software structure [6], or to output SR as an open textual format[7].

In accordance with [2],"Generally speaking, API is a universal way to implement any required functionality, however with changing requirements it's impossible to predict a spectrum of clients' needs".
And an open SP format can be a solution to potential problems: "open formats usually have a lot of access means: from simple APIs to high level specialized products. Besides, it's possible to implement one's own interfaces to process SR represented with open format".

A particular format may be something self-designed[7] or a standardized solution such as XML[8], or JSON[9], as an alternative.

1.3 LSP and distributed approach to building development environment

Considering the things discussed above, nowadays we have a solid basis to provide a good tooling based on semantic analysis: methods to represent software source code‘s Semantic Representation and evaluate it accordingly to clients‘ needs.

Modern IDEs apply those methods to deliver a decent service, but still there is a problem: those software products use their own implementations of compilers, usually proprietary and unrelated to the original language‘s development team. It implies a set of problems noted in the 1.1.

Having a good modern compiler capable of generating an SR makes things a bit less complicated but still doesn‘t solve the language-specific IDE implementation time and cost problem.

Obviously, these problems are not unique for the IDE class of products, but for any big monolithic architecture, and the solution may be pretty straightforward: if we can lower the bonding of the system and represent a development environment as a set of tools instead of one integrated solution, we can distribute the IDE implementation to have a set of disjointed modules:

- An editor
- Compiler to SR
- SR clients (described in 1.1)
- Protocol between an editor and the language-specific part

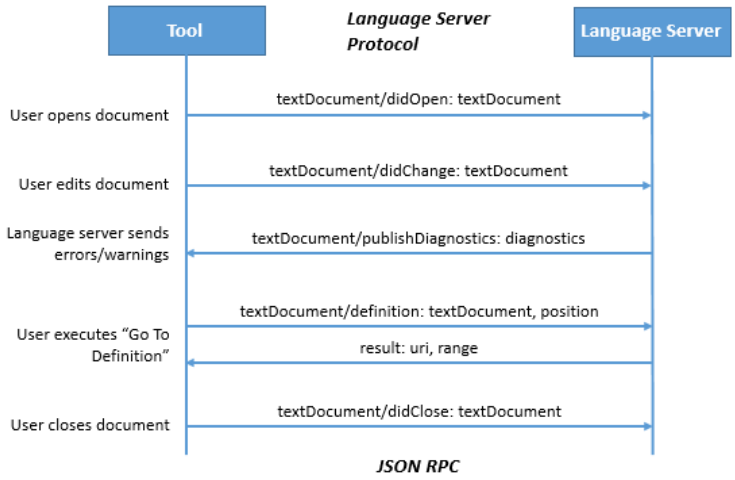

Figure 1.1: Language Server Protocol

The last one have not been introduced yet: the protocol to connect any third-party tool to the language infrastructure, achieving a decent IDE-like functionality without its maintenance and development costs

Language Server and Language Server Protocol introduced by Microsoft in 2016 represent a development environment as two disjoint parties:

- Language Servers to implement all the SR analysis things
- Clients as editors or other development tools using the LSP to communicate with Language Servers [10]

1.4 Conclusion

Conventional compilers with a monolithic architecture, that are only good at executable code generation, are hard to integrate into a modern development environment as they do not share semantic representation of the source code, thus to develop a good IDE one must write their own source code to Semantic Representation compiler.

A modern compiler (that does share a high-level intermediate representation) is a big step towards simpler language toolings and it can become even more convenient combined with a distributed IDE architecture that splits an editor and the language toolchain into two disjoint parts, linking them via a standardized protocol.

This approach gives language developers a great opportunity to make use of an existing development infrastructure, providing their Language Server for a giant set of development tools, as well as a way to fearlessly experiment with new and existing analysis techniques, e.g. a Software Knowledge Base[11], described by Bertrand Meyer, may be implemented as a language server module, as an alternative approach to the one selected by the original author in 1985: integration of analysis tool into an editor was not possible back then, but this is the exact thing that LSP is good for now.

Concluding, the Language Server may be considered to be the most feasible solution to rapidly bootstrap rich development infrastructure for aspiring new languages, with a broad path to evolve further.

Chapter 2

Methodology and architecture

2.1 Compiler integration

The Language Server idea is to launch the LS instance in the same project directory opened in the editor, and connect it to the editor via Language Server Protocol.

A Language Server is responsible for language-specific editor features, it works on the language Semantic Representation and other metadata to perform semantic analysis and consequently provide the editor with usable data in the agreed format via Language Server Protocol. As Language server heavily relies on the modern compiler, that exposes the SR, we need to implement a way to integrate compiler into the Language Server and to enable their inter-operation.

There are two possible ways to achieve that: either use the compiler as a library or invoke it in a separate process, feeding specific command line arguments.2.1

invoking as a command	**using compiler as a library**
simpler integration	harder integration
very limited invocation options	complex invocation strategies may be expressed
need to (de)serialize data	can exchange binary data
need to implement IR traversal in the LS	compiler can expose AST traversal API
need to describe compiler internal data types in the LS	compiler can expose internal data types

Table 2.1: Compiler integration methods comparison

Since the SLang[1] compiler does not expose any AST traversal API or internal

data types, most of the traits specific to an “integration as a library” option will not be used in our case. Moreover, the compiler provides a stable JSON-formatted SR, which, being a text-serialized format, can be easily transferred via an operating system channel like standard output[12].

This way, we get a number of convenient features: the compiler can be invoked by our Language Server as a command call,this option is easier to implement on both Language Server and compiler sides. And since we are not limited with any functionality that would require “compiler as a library” traits, we can declare this way of integration the most feasible in our case and stick to it.2.1

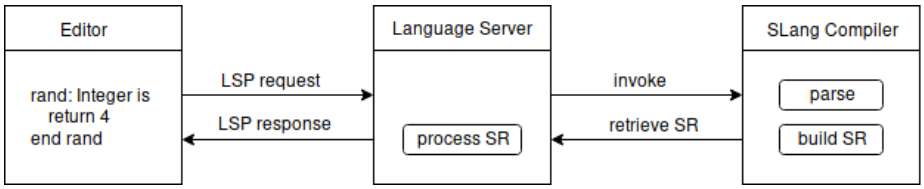

Figure 2.1: Workflow with Language Server and compiler

2.2 Language Server Extensible Architecture

The main idea of this research is to bring architecture of Language Servers to the next level, make it modular and extensible, thus allowing third parties to throw in additional functionality for the SLang tooling with no need to hack into the Language Server code.

Therefore, the architecture of the SLang Language Server is divided into two aggregate components: 2.2

- The Core
- Module System

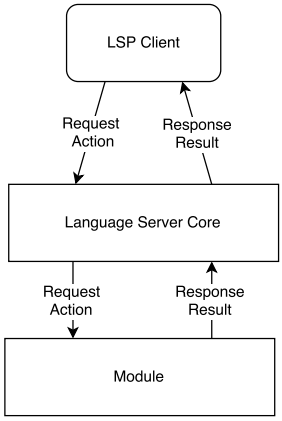

Figure 2.2: Language Server High-level Architecture

2.2.1 Language Server Core

Language Server Core is a basement level of the Language Server on which Language Server Modules will operate. Responsibilities of LS Core include:

- LS Client connection maintenance
- Module registry maintenance
- Routing of incoming requests and data control flow between modules

Each of these responsibilities we shall describe in detail below.

Client connection maintenance

According to the Language Server Protocol[10], the client controls the lifetime of a server, i.e starts it and shuts the server down on demand. After startup, the client connects to the server using one of the transports. Since the transport level is not constrained by the LSP, specific transport can vary in different implementations.

Language Server Core should support several types of transport and be able to operate on them to accept requests and respond to the client. The list of widely used transports we are going to implement is

- stdin/stdout
- TCP
- UDP

Implementing that list will supply the most of LSP clients with an option of how to work with the SLang Language Server.

Module Registry

To be a foundation for an extensible modular architecture Language Server Core needs to have a subsystem for module registering, maintenance and interoperation organization.

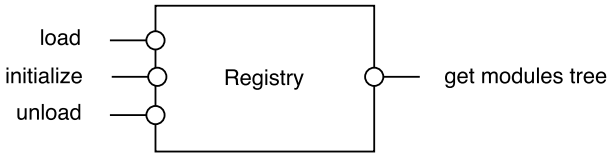

Figure 2.3: Module Registry API

The registry should register a module and share its status, as well as information on how to launch and connect internal (core) and external modules. On startup, the registry should initialize its state using a predefined directory containing configuration files. Afterwards, it should maintain an API to load and unload additional modules via LSP. Consequently, we need to extend the LSP with additional commands for the modules registry:2.3

- registryCtl/load: register the module.
- registryCtl/unload: unregister the module.
- registryCtl/status: query module status.

Dispatcher and data flow control

After startup, connection setup, module registering and initialization, the Language Server accepts the first request from the client. This request gets validated by the Language Server Core, and then, after looking up the Module Registry, the request gets handed over to the beginning of a processing pipeline responsible for handling this type of requests.

The dispatcher part is basically a glue, that connects all Language Server components together and maintains the data flow edges of the modules graph, enabling module inter-operation by the rules loaded into the Registry, that are discussed further in the section 2.2.2.

There is a simpler alternative approach to module inter-operation organization: let the modules send data to each other and organize pipeline as they want. Although this peer-to-peer schema here would save a lot of bandwidth, it would also inevitably lead to the dependency hell, as such an approach would require having every module knowing about each other and to be connected to each other. Thus, here we face the classic client/server trade-off: we can offload a "server" (LS Core) only if we complicate a "client" (modules). Since the client side is to be developed by third parties, the simpler it is – the better: the server, controlling all the data flow, will leave the module developer only with the business logic implementation tasks.

2.2.2 Module System

Language Server is a great idea that enables IDE-like functionality for comparably simple text editors, but currently these are mostly designed as a monolithic software, while their service functions are naturally extensible, e.g. it is common for a static analyzer to have modules for different diagnostics, and similarly one of Language Server functions is to provide an editor with diagnostics, so here we can have a hierarchy of at least two modules – the compiler diagnostics and the diagnostics provided by an external static analysis tool. We can go even

further and implement a static analyzer as a combination of the base static analysis module with a bunch of atomic diagnostic modules derived from this base module. The same logic is applicable to every Language Server service to some extent.

Separation of the monolithic core from the actual functionality implemented through modules will significantly lower the Language Server internal bonding, and will make module implementation relying on as stable API as possible, therefore allowing the core and modules development to be performed simultaneously with no mutual API breakages.

Therefore another benefit of an extensible architecture that we can derive is the Language Server easy adoption for any specific corporate needs. As implementing some additional functionality would be as easy as developing a plug-in using the Language Server Modules API, corporate users can extend the Language Server according to their specific requirements, i.e add custom diagnostics, enforce code style, or even hook-up the proprietary static analysis or code generation tools.

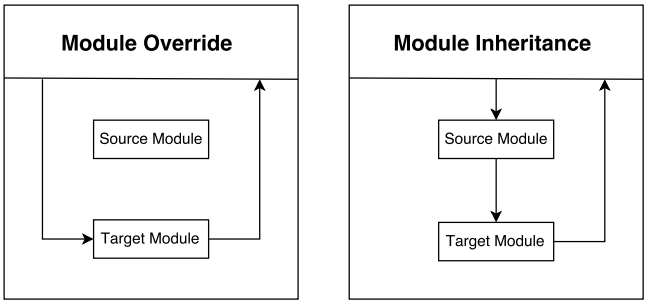

Figure 2.4: Module Hierarchy

One of the ways to develop such architecture has already been introduced above: the hierarchy of modules expressed with the basic Object-Oriented Programming terms2.4. In this hierarchy, modules can either override other modules or extend them in a way of post-processing the results of the base module computation. So the modules would form a classic tree, deriving from a base module, and intercepting the data flow. The example of such a module hierarchy is illustrated in the figure 2.5

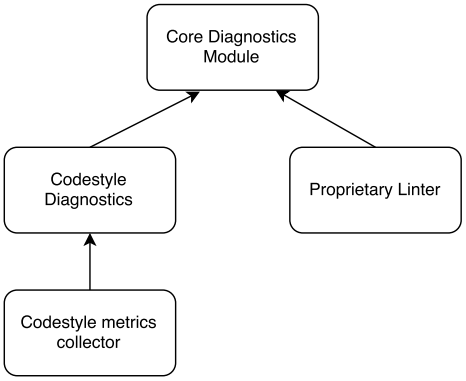

Figure 2.5: Example Modules Tree

We already mentioned the hierarchy of the base and derived modules above. Taking the modern Object-Oriented languages' type systems as an example, we can build a system with the base modules, which are to be deployed bundled with the Language Server and responsible for one particular LS Protocol method each. These modules will mainly provide the basic runtime for all the derived modules:

- set up data types: the input information, analysis context, and the results.
- perform the construction of the context (if applicable).
- intercept analysis results of the derived module and send them to the client through LSP.

As one could have figured, it is not really a conventional O-O paradigm inheritance model: core modules do not let the derived ones to override their logic as they work with the Language Server Client and therefore are forced to live within the same process with the Language Server due to resources sharing. The process of a modular Language Server operation is showed on the sequence diagram 2.6.

Hereby, we have two types of modules:

- Core Modules: embedded into the LS process, exposing the run-time for external modules, cannot be overridden.
- External Modules: pluggable things, derived from the core modules, can be overridden.

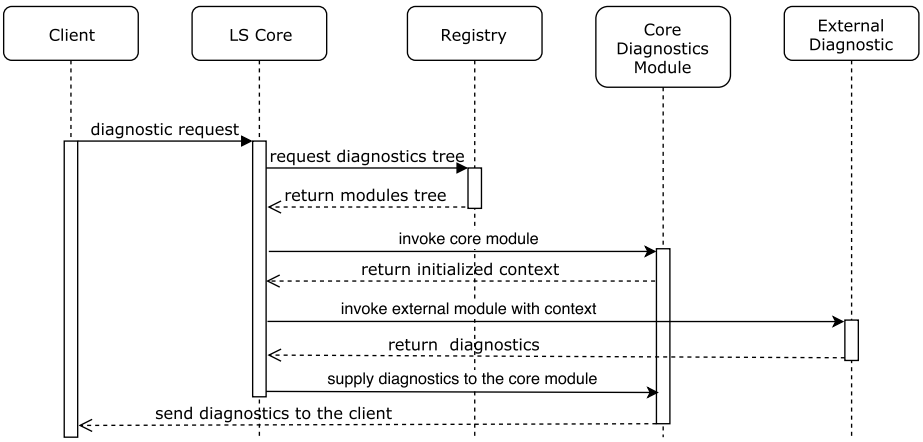

Figure 2.6: Modules invocation sequence diagram

Summing up, now we have a way to integrate the compiler into the Language Server, powerful extensible architecture that allows to throw in additional functionality for the Language Server, and modules hierarchy to start designing the core modules for the Language Server Protocol methods, as well as the external ones to perform actual analysis and supply Language Server Clients with the source code insights.

2.3 Development Plan

To implement a complex system it is vital to split the work into phases and introduce the iterative development plan, that would cover a small self-contained part of work on each iteration, in such a way that the result of each iteration could be presented as a working product.

Also, each iteration should include unit and integration testing to form good test coverage at the end of the development and to simplify the process of project evolution through use of regression testing.

A Dummy Language Server
At first, the very basic functionality should be implemented: accepting the connection and handling a request with a hardcoded response

SR-driven Language Server
The next step would be to integrate the compiler and the language Semantic Representation to feature “jump to definition” for the SLang Language Server.

Basic Modular Language Server
After all the groundwork with the protocol and compiler inter-operation, modu-

lar architecture implementation may be started: without foreign process extensions for now, including only built-in modules, that operate in the same address space with the Language Server, and the simple registry for them, to mock the basement for the future extensible LS.

Extensible Language Server

Finally, the last step is to extend the registry so it would handle configuration files and load external modules and to extend the inner protocol and API to work with modules that are operating in external processes, thus completing the extensible Language Server implementation for SLang. The final architecture after this stage is illustrated in the figure 2.7

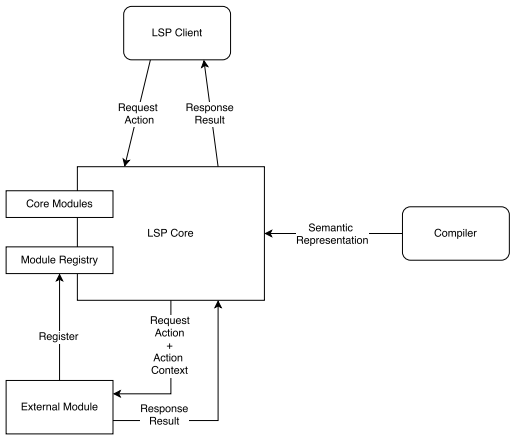

Figure 2.7: Final architecture

2.4 Approaches to implementations

The most of the Language Server functionality heavily relies on the language‘s semantic representation. For the first prototype we are going to be using the SLang semantic representation and will build a few tools for it.

The SLang is using a json-formated SR with an object hierarchy described in the following section.

2.4.1 SLang Semantic Representation format

Slang has three scoping components: Unit, Block and Routine, where any nesting order is valid. Each of these components has its context with encapsulated members as an array to keep the consistent order:

```
{
    "type": "Unit",
    "context": [
        {
            "type": "Routine",
            "context": [
                "type": "Block",
                "context": [
                    ...
                ]
            ]
        }
    ]
}
```

In addition to the described basic scoping hierarchy, SLang JSON SR unilizes some extra fields with metadata and semantics extracted from the source code:

- **name**: the program-specified name of a specific program entity (variable, routine, etc.), optional.
- **id**: unique entity identifier, used for SR internal referencing.
- **ref**: internal SR reference to an **id**, used for referencing the parent unit, variable definitions, etc.
- **span**: text locator of the entity containing the start and the end positions represented as a line:column pair.
- **signature**: the routine special case field containing the routine signature as input and output contexts.

The meaning of each of these fields is implementation-defined and their semantics rely on the entity type they are contained within. For instance, for a **Unit** type, the **ref** field will be pointing to a parent **Unit**, and for a **Var** within an **Expr** entity the **ref** field will be a reference to this particular variable definition. Thus we can avoid lengthy SR leaf definition in the Language Server code saving all the important semantics.

2.4.2 Jump to definition

The most basic developer tool as well as the most basic toolset part is a **jump to definition** command. Basically it helps the developer to instantly open the file and the line with the definition of some code entity used in the source code.

To implement this part of the Language Server via analysis of SR, only three of its components are necessary:

- **id**: every definition inherently has its own unique identifier.
- **ref**: every usage of an entity defined somewhere references the original definition by its id.
- **span**: knowing the id, the location of the definition can be found.

2.4.3 Code completion

Another very common instrument in a developer toolbox is autocompletion, provided by an IDE. In our case the provision of an autocompletion is on the Language Server.

The most usual case of autocompletion usage is navigation among the encapsulated class (or **Unit**) members, where autocompletion can provide user with the useful suggestions on the names of these members and their types. Using the recursive **jump to definition** search, we can find a scope with the members to be suggested, read theit metadata and present it to a user in a readable format.

Chapter 3

Implementation

The project was decided to be named Akkadia, trivia of this name can be found in Appendix A 6.1. This chapter describes implementation specifics of Akkadia.

3.1 Instruments

3.1.1 Programming Language

As the language with which researcher is most acquiented with is Rust, a modern programming language that aims to provide zero-cost abstractions, memory and thread safety and blazing speed, with a decent coding experience at the same time, it was decided to implement this work in it. Becides an author's favor, Rust can provide the following features that are useful for a prototype ought to be implemented in a limited amount of time:

- Rich type system for complex abstractions and code reuse
- Compile-time error catching for reduced debugging time compared to the other non-GC languages
- Deterministic resource management through RAII
- Convenient error handling through sum types that solves the problem of raising unextected (and, hence, unhandled) exceptions from foreigh codebase, because errors are returned from function as a part of Result enum, and at the same time makes returning errors practical compared to C/C++ approach. In virtue of Result concept, user of a function that can fail can not use the data avoiding error handling. Thus Rust helps to avoid problems which are inflicting noticeable damage caused by hard-to-debug issues such as a Null Pointer Exception in non-null-safe languages or the use of uninitialized memory.

3.1.2 IDE

The main part of a developers tooling nowadays is an IDE capable of providing code analysis, hints, refactoring methods, and other integrated unilities to speed up development process. For an author there were a choice between two editors/IDEs: **Visual Studio Code** and **Intellij Idea** with respective plugins for the Rust language support. Both of them had their pros and cons at the moment of writing this paper:

Visual Studio Code	Intellij Idea
small response times, lower memory footprint	bit slow, needs a powerful workstation
bad support of Cargo Workspace project	supports everything that can be built with Cargo
poor code analysis	type info and linter through plugin
good cargo integration for the whole project	can only show errors in the current file
stagnation in plugins development	active plugin development

Table 3.1: Code editors comparison

The issues of both code editing solutions are to be solved by Rust Language Server (with adding some more problems), which is currentry under heavy development and on the oppinion of the author is not ready to be integrated into the everyday development practice.

3.1.3 Donor codebase

To not to get drown under tons of work, it was decided to choose an existing open-source language server as an implementational basis.

Rust ecosystem have to provide an open-source language server written in Rust for Rust, which can be forked and adopted for the respective use in SLang ecosystem. RLS (Rust Language Server) has a well-tested architectural basis to employ as a necessary basement level, which would be a long work to implement and debug otherwise. Parts of RLS which are useful for the SLS implementation:

- Language Server Protocol implementation
- VFS(Virtual File System)
- Convenient API to accept and dispatch Language Server messages

3.2 Preparing RLS codebase

The first step of implemeenting Slang Language Sever using RLS as a basis is to prune all Rust language-specific functionality, that is connected to the Rust compiler or other Rust code analysis tools, such as the **racer** tool, responsible for autocomplete hints. That reduced the code base almost in a half, leaving only Language Server service code and placeholders for future functions implementation.

After the first step, the project ended up with the following structure:

- **akkadia**: the main language server source tree:
  - **actions**: control messages definitons and corresponding actions implementations:
    - **notifications**: messages that do not require response
    - **requests**: messages that do require response
  - **server**: io and the main server loop
    - **service**: main loop and a glue to connect all components
    - **io**: input-output handling
  - **test**: integration testing
- **akkadia-span**: types for identifying code spans/ranges
  - **compiler**: conversion from the compiler span/range types to the Akkadia types
  - **main**: type definitions
- **akkadia-vfs**: virtual file system to keep file tree and file cache

3.3 Module System

Akkadia is designed to be extensible with external (out of source tree) modules, so the inter-module IPC should be language-agnostic, therefore the most easy-to-employ and probably the most flexible solution is to use serialized data on the input and the output of the module interface.

Defined in Rust types, the **Input** and **Output** of a module looks as follows:

```
// A trait defining module parameters
pub trait Params: serde::Serialize + serde::Deserialize<'static> {}
// A trait defining module response data
pub trait Response: serde::Serialize + serde::Deserialize<'static> {}
```

This basically means that anything that can be serialized and deserialized in an arbitary format supported by serde-rs**??**, can be a module argument or a module return type. These are just convenience flag-traits, that do now provide their own methods.

Consequetively, there should be a data type that implements one of those traits (or both). In the current prototype, all modules have full access to the Language Server state, so it is provided to them via IPC interface, in serialized data format:

```
#[derive(Serialize, Deserialize, Debug, Clone)]
pub struct State {
    // Virtual File System containing the source tree
    vfs: Vfs,
    // Current project path
    path: PathBuf,
    // Method on which the module was invoked
    method: Method,
    // A mutable responce list
    responses: Vec<Response>
}

impl Params for State {}
impl Response for State {}
```

The first fields are self-explanatory, but the method and response types are a bit complicated. To start with the **Method** type, it is to be said that it is a enumeration containing different method names as the data supplied with the method.

```
pub enum Method {
    Completion(Span),
    <...>
    ExitNotification
}
```

Method itself is an aggregation of both **Notifications** and **Requests**. Intuitively, the only difference is that the first ones do not require **Response**. **Response** is a enum with variants containing data too, the specific data structures that are expected to be returned by the respective methods.

```
pub enum Response {
    Completion(Hint),
    <...>
}
```

All of that is used in the **Module** trait which defines the necessary IO methods for the IPC communication to the modules.

```
pub trait Module {
```

```
    type Params: Params;
    type Response: Response;
    type Error;
    fn call(params: Self::Params) -> Result<Self::Response, Self::Error>;
}
```

This is a very generic trait that can be implemented for basically any module IPC IO needs, be it basic stdin-stdout, TCP socket connection or even some wicked blockchain-based external service. The specific implementations of the **Module** trait define the built-in module types that can be loaded via config file and used with no Language Server code hacking necessary. This built-ins are not easily extendable, but extendability is possible indeed through shared libraries, for example, as the users of the Module trait inside a Langage Server are not aware of the concrete types because of extensible use of **Box¡Trait¿** pattern, that provides dynamic dispatch capabilities for any types implementing the **Method** trait.

To summarize modules data flow, the implementation of a very basic language server module that accepts json-formatted state, reacts to a Completion method and returns a ”hello world” hint would be

```
extern crate serde;
extern crate serde_json as json;
extern crate akkadia;

use std::io::{
    stdin,
    stdout,
};

use akkadia::{
    State,
    Method,
    Response,
    Hint,
};

fn main() {
    // Read serialized State
    let mut state: State = json::from_reader(stdin())
        .unwrap();

    // Add the "Hello World" hint to responses
    match *state.method {
        Method::Completion(span) => {
            state.responses.push(Hint::new(span, "Hello World!"))
        },
        _ => () /* do nothing */
    }

    // Write State back to the stdout
    json::to_writer(stdout(), &state)
        .unwrap();
}
```

3.4 Dispatcher and the Module Registry

The core component of the module system that is accountable for routing data to the modules is the **Dispatcher**. It works in close communication with the **Module Registry**, a part of system responsible for holding a list of modules with their characteristics, loading lists of modules and building the chains of

modules.

Firsty, when the Language Server accepts a request, it invokes the **Dispatcher**

```
impl Server {
    fn loop(mut self) {
        loop {
            if let Err(e) = self.step() {
                eprintln!("{}", e);
            }
        }
    }

    fn step(&mut self) -> Result<(), Error> {
        let message = read_message(&mut self.input)?;
        let state = self.dispatcher.handle(message)?;
        Ok(write_results(&mut self.output, state.responses))
    }
}
```

The dispatcher is connected to the **Module Registry** that is able to supply all the necessary information about modules and their sequences.

```
struct Dispatcher {
    registry: Registry,
}

struct Registry {
    modules: Vec<Box<Module>>
}
```

As it can be recalled from the previous section, modules may influence each other in 2 ways: override and build composite-like chained structures. In the first case, an override is transparent for the **Dispatcher**, as the overriden modules get unregistered inside the **Module Registry**. The later case is not though, so it is a place to introduce the **Module Chain** abstraction, which is basically a Rust iterator over modules:

```
impl Registry {
    // Returns an iterator over module chains
    fn get_method_handlers(method: &Method) -> impl Iterator<Item=impl Iterator<&Meth
        /* build method handlers chains based on inheritance rules */
    }
}
```

Once the chain list is built, **Dispatcher** uses that info to run the modules:

```
impl Dispatcher {
    // Returns an iterator over module chains
```

```
    fn handle(&self, message: Message) -> Result<State, Error> {
        let method = message.into_method();

        let empty_state = self.make_empty_state(&method);
        let final_state = empty_state.clone();

        for chain in self.registry.get_method_handlers(method) {
            let chain_state = empty_state.clone()
            for module in chain {
                chain_state = module.run(chain_state)?;
            }
            final_state.merge(&chain_state)
        }

        Ok(final_state)
    }
}
```

So, the dispatcher has two kinds of state handling: it merges state of indepentent chains and applies a replacement technique for the state that is propagated through a single chain. At the same time, all the chains accept the initial state unchanged and different chains can not affect each others data.

3.5 Autocomplete

Autocomplete is the most crucial and most used development helper, that is crucial for any code editor to have. It is a good self-contained example of the Language Server module implementation, that employs both the module system and the SLang compilers Semantic Representation of the source tree.

The Semantic Representation of a language contains the source code entities such as **Units**, **Routines** and **Variables**, as well as the scope information, that can be used for autocompletion. Besides the dynamic entities, that are different from codebase to codebase, the language itself has **keywords**, which are to be autocompleted too.

Therefore the autocompletion high-level algorithm will look as follows:

- Accept the user input
- Find matches in keyword list
- Find matches in entities in the current and parent scopes
- Build a list of hints
- Return to the user

Matches search is a straightforward task, but it might be quite slow on big code bases. There are several methods to approach the speed problem, both on compiler interface and on a matches search ends. First of all, it is not feasible to re-run parser every time user might need an autocompletion hint, so the parser output shoud be cached and updated as a background task that would not block the hint generation process. This way the parser re-runs should not be noticeable by user at a level beyond a small lag in completeness of autocomplete hints list.

The next method approaching the matching speed includes extensive use of an indexed cache and an appropriate algorithm for the matching task, particularly an **Aho-Corasick** algorithm that is known to be used in several autocompletion tools, such as Rust‘s **racer**.

Once built, the Aho-Corasick index may be used until the next change in the source tree to perform rapid searches. Aho-Corasick automaton is implemented in rust crate **aho-corasick**, that is imported in the autocompletion module implementation.

3.6 Language Server control utility library

Testing and Language Server usage as utility outside from within of client-editor Language Server Protocol client would require a CLI tool, that would be able to act like and aditor and provide an appropriate interface to run single or batch tasks.

Basically, that is a simple language server client that can connect to the Akkadia and then perform the messages exchange via jsonrpc like a real Language Server Client in the editor would do. It have been imlemented for integration testing of the Language Server.

Chapter 4

Evaluation and Discussion

The implementation of Akkadia is compliant with **jsonrpc** standard used in the Language Server Protocol as it inherits the implementation from Rust Language Server.

Language Server Protocol includes a list of request methods that Language Server can or should handle. Akkadia Core has an implementation to handle file notifications to update its internal Virtual File System and the mandatory service methods:

- ExitNotification
- Initialized
- DidOpen
- DidChange
- Cancel
- DidSave
- DidChangeWatchedFiles
- ShutdownRequest
- InitializeRequest

The other methods suck as the implemented **Completion** and are to be impelemted as the connectable **modules**. Module System was implemented to fully support the modular approach described in the Methodology, except the configuration files and the on-the-fly module integration.

The other Language Server methods responsible of extending code analysis capabilities of an editor are to be implemented as a further work, namely

- DidChangeConfiguration
- GoToDefinition

- ShowReferences
- Rename
- FindImpementations,
- ShowSymbols
- Formatting,
- RangeFormatting,

This can be done in any language that supports JSON encoding.

Also, continuing on module system, it may be extended by providing more auxillary data to the modules, via allowing them to have an internal storage in the **State** structure, to improve dependent (via inheritance) modules interoperation capabilities. This way Akkadia could support many languages as it was planned initially, via providing the 'Core' language modules, responsible for compiler integration, that would store the Semantic Representation in this additional storage, for the use of the modules that inherit from the core language module.

Considering the speed of execution, the current 'naive' implementation of the module IPC may be improved through the use of shared memory where it is appropriate. This way there wouldn't be performance losses on transmitting huge data structures such as the Virtual File System.

The major achievement of the Akkadia architecture and implementation is an easy to use **Module System**, that is extendable without any Rust or any other language-specific instruments, libraries, or input-output method limitations hence is providing great flexibility.

Chapter 5

Conclusion

Language Server is a new way to bring modularity and extensive code reuse into modern integrated development toolsets, that employs language's compiler to perform code analysis.

With the architecture described in this thesis, this modularity may be brought to the next level, allowing to inject virtually any analysis, statistical, or other tools into the Language Server, making it possible to extend simple code editors up to a point of building very powerful toolsets. Also this approach to building a Language Server can make the concept of LS agile enough to make it an easy to use instrument for enterprise users, who are tending to have a lot of specific internal tools, which may be now incorporated into the Language Server itself.

From the perspective of a programming language developer, Language Server allows to rapidly bootstrap a ready-to-use environment of different code editors supporting the inspection tools for this new language.

The Language Server approach allows to reduce time and cost of development of the language support in major editors and IDEs at least by the cost of implementing a parser, as it is reused from the compiler. Additionaly one Language Server implementation can support a variety of different editors, while a plugin is usually able to provice a language support only for a single editor or IDE.

Concluding, the Language Server may be considered to be the most feasible solution to make a rich development infrastructure for aspiring new languages, with a broad path to evolve further through the introduced module system.

Bibliography

[1] Eugene Zouev and Alexy Kanatov. “SLang Programming Language”. 2017.

[2] Eugene Zouev. “Evolution of Compiler Architecture”. In: *ISBN 5-317-01413-1* (2005), pp. 322–331.

[3] Eugene Zouev. “Semantic APIs for Programming Languages”. In: (2010).

[4] Free Software Foundation. *Ada Compiler GNAT*. 2016. URL: http://www.adacore.com/home/products/gnatpro/ (visited on 09/30/2017).

[5] Aemon Cannon. *Building an IDE with the Scala Presentation Compiler*. URL: http://ensime.blogspot.ru/2010/08/building-ide-with-scala-presentation.html.

[6] Mark Linton. *Queries and Views of Programs Using a Relational Database System*. 1983.

[7] The Rust Team. *Rust Programming Language Blog: MIR*. 2016. URL: https://blog.rust-lang.org/2016/04/19/MIR.html (visited on 09/30/2017).

[8] R. Germon. *Using xml as an intermediate form for compiler development*.

[9] ECMA-404. “The JSON Data Interchange Format”. In: *ECMA International* 1st Editio.October (2013), p. 8. ISSN: 2070-1721. DOI: 10.17487/rfc7158. arXiv: arXiv:1011.1669v3. URL: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf.

[10] Sourcegraph. *A community-driven source of knowledge for Language Server Protocol implementations*. URL: http://langserver.org/ (visited on 09/30/2017).

[11] Yuan Wanghong et al. “C++ program information database for analysis tools”. In: *Proceedings Technology of Object-Oriented Languages. TOOLS 27 (Cat. No.98EX224)* (), pp. 173–180. DOI: 10.1109/TOOLS.1998.713598. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=713598.

[12] The Open Group. *Standard I/O Streams*. 1997. URL: http://pubs.opengroup.org/onlinepubs/7908799/xsh/stdio.html.

Chapter 6

Appendix A: Trivia

6.1 Akkadia: origin of the name

Acknowledgment's

I want to thank my thesis supervisor Eugene Zouev for his support, guidance, the open minded approach to my ideas, and for helping me to shape the concept of this work. Thanks to all participants of SLang project, for your outstanding work, creative thinking and constructive criticism, namely Semyon Bessonov, Andrew Ermak, Vadim Gilemzjanov and Ilfat Mindubaev.

Special thanks to my dear friend Alena Yuryeva for helping me to not to fall into depths of procrastination and for proof-reading of the thesis text.

Thanks to Rust Comminuty for helping finding the answers I needed, and for joyful chat holywars about the thesis subject. Thants to Rust Team and the authors of other tools I happened to use to implement Akkadia. Thanks to authors of papars I referenced in this text. Thanks to Innopolis University.

And thanks to my supporting family who gave me an opportunity to become who I an now for me to write this thesis.