Language Server implementation for SLang

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

Language Server implementation for SLang

Contents

Abstract

abstract ...

Chapter 1

Introduction

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 monolithyc archithecture, usually standalone from the language compiler infastructure 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 archithectured 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 multiparadigm SLang programming language.

Chapter 2

Literature Review

2.1 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‘” [1], 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[1]. To build it one needs to reimplement 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[1, 2], 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[3]. 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++[2], 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.

2.2 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 [1, 2], 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 [3, 4], relational database representing a software structure [5], or to output SR as an open textual format[6].

In accordance to [1],"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[7], or JSON[8], as an altetnative.

2.3 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 2.1.

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 2.1)
- Protocol between an editor and the language-specific part

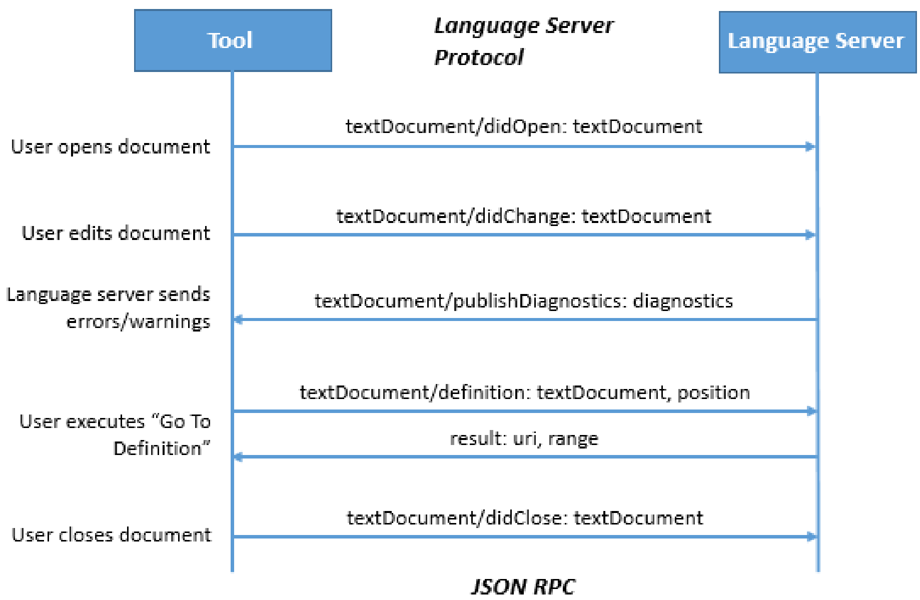

Figure 2.1: Language Server Protocol

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 devtools using the LSP to communicate with Language Servers [9]

2.4 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[10], 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 3

Methodology

3.1 SLand Semantic REpresentation Design

Waiting for the SR meeting results

3.2 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 interoperation.

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

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 3.1: Compiler integration methods comparison

Since the SLang[TODO] 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 transfered via operating system channel like standard output[TODO].

Thus the compiler can be invoked by our Language Server as a command call, we are not limited with any functionality that would require "compiler as a library" traits, and this option is easier to implement on both Language Server and compiler ends, therefore we can declare this way of integration the most feasible in our case and stick to it.

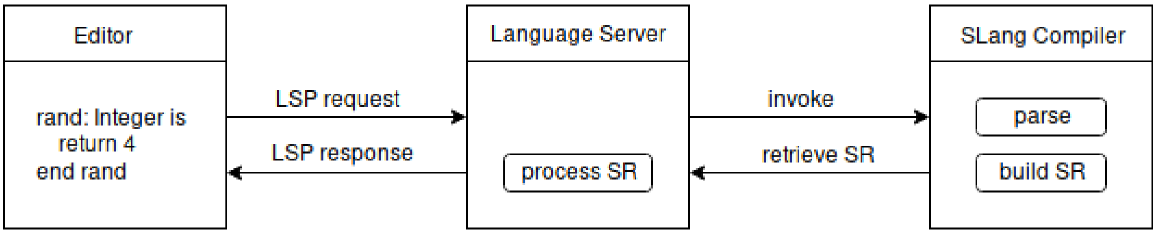

Figure 3.1: Workflow with Language Server and integrated compiler

3.3 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
- Dispatcherisation of incoming requests and data control flow between modules

Each of these responsibilities we shall describe in detail.

3.3.1 Client connection maintenance

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

Language Server Core should support several transports and be able to work operate on them to accept requests and respond to the client. The list of widely used transports we will 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.

3.3.2 Module Registry

For the extensible modular architecture described in **??** Language Server Core needs to have a subsystem for modules registering, maintenance and their interoperation organization.

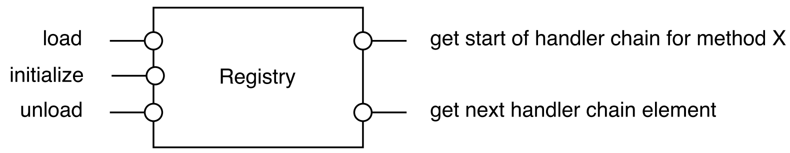

Figure 3.2: Module Regustry API

The registry should register and give module status, as well as information how to launch and connect internal (core) and external modules. On startup, registry should initialize it's 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 an additional commands for the modules registry:

- registryCtl/load
- registryCtl/unload
- registryCtl/status

3.3.3 The Dispatcher

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

Basically the dispatcher part is a glue, that connects all Language Server components together and maintains the data flow "edges" for the modules graph, enabling module interoperation by the rules loaded by the Registry and that are discussed further in the section [TODO].

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

3.4 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 the Language Server code.

3.4.1 Code Semantic Based Highlights Design

Architecture and methods of IR analysis for hightlights

Mapping from semantic entities to LSP highlighting staff

3.4.2 Autocomplete Design

Description if autocomplete argorithms and data structures choice

Chapter 4

Implementation

4.1 IR Design

Design desicions behind SLang IR implementation, its structure and examples

4.2 Language Server Design

Design desicions begind language Server, its architecture

4.3 Language Server Implementation

4.3.1 Compiler interoperation

Implementation of compiler interoperation in language server

4.3.2 LSP implementation

Implementation of Language Server Protocol

4.4 LS Modules

Description of basic Language Server module design and implementation

4.4.1 Semantic Based Highlights

Description of the semantic based highlights

4.4.2 Autocomplete

Description if autocomplete implementation

4.4.3 Documentation Generator

Description of documentation generation implementation

4.5 Language Server control utility library

Description of C ABI LS control library API

Chapter 5

Evaluation and Discussion

Recap of thesis subject

LSP standard complainance testing

Performance testing

Estimation of developement cost of the full-featured IDE sor SLang, comparing that with time and cost of Language Server Implementation

Discussion of future developemnt opportunitits

...

Chapter 6

Conclusion

Recap

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.

. . .

Bibliography

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

[2] Eugene Zouev. "Semantic APIs for Programming Languages". In: (2010).

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

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

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

[6] 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).

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

[8] 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.

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

[10] 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.