Title

Federico Cristiano Bruzzone
Id. Number: 27427A

MSc in Computer Science

Advisor: Prof. Walter Cazzola
Co-Advisor: Dr. Luca Favalli

UNIVERSITÀ DEGLI STUDI DI MILANO
Computer Science Department
ADAPT-Lab

Academic Year 2023-2024

COMPUTER SCIENCE DEPARTMENT • ADAPT-LAB

Contents

1
Introduction

2 Background

In this chapter, we provide an overview of the concepts and technologies that are relevant to the work presented in this thesis. We start by introducing the concept of language servers and the Language Server Protocol (LSP) in Section 2.1. We then discuss language workbenches in Section **??**, type systems in Section **??**, and software and language product lines in Section **??**.

The goal of this chapter is to provide the reader with the necessary background knowledge to understand the work presented in the following chapters. We assume that the reader has a basic understanding of programming languages and software development.

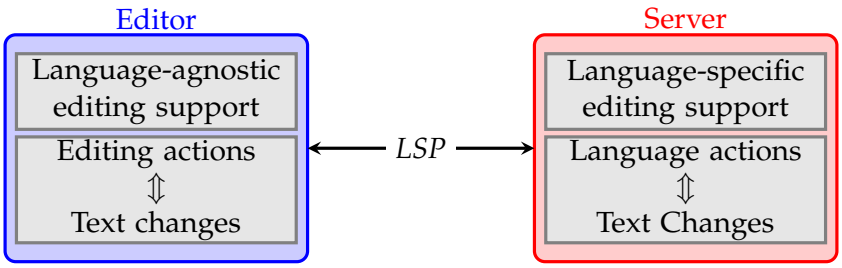
2.1 Language Server Protocol

The Language Server Protocol[1] (LSP) is a protocol that allows for the communication between a language server and an editor or an IDE. The LSP is used to decuple the a language-agnotic editor or integrated development environment (IDE) from the language-specific features of a language server (see Listing 2.1). This allows for the development of language servers that can be used with multiple editors or IDEs. The LSP is based on the stateless JSON-RPC protocol and defines a set of messages that are used to communicate between the language server and the editor or IDE.

Usually, the LSP Clients are developed as plugins for popular editors or IDEs decresing the effort to support a new language in a given editor. The LSP Clients are responsible for sending requests to the language server and processing the responses. The language server is responsible for providing language-specific features. The LSP defines a set of messages that are used to communicate between the language server and the editor or IDE. These messages include requests for code completion, code navigation, and code analysis, as well as notifications for changes to the document, diagnostics, and progress reports.

Language servers are *de facto* standard for providing language-specific features in editors and IDEs. The LSP is supported by a wide range of editors and IDEs, including Noevim[2], Visual Studio Code[3], Eclipse[4], and IntelliJ IDEA[5]. There are several language

[1] https://microsoft.github.io/language-server-protocol
[2] https://neovim.io/doc/user/lsp.html
[3] https://code.visualstudio.com/api/language-extensions/language-server-extension-guide
[4] https://www.eclipse.org/community/eclipse_newsletter/2017/may/article1.php
[5] https://plugins.jetbrains.com/docs/intellij/language-server-protocol.html



Listing 2.1. *LSP approach to language support. Borrowed from [26].*

servers available for popular programming languages, including Rust, TypeScript, Python, and Java and most of them are open-source[6].

The LSP is initiated by Microsoft and is now an open standard that is maintained by the Language Server Protocol Working Group. It was designed for the use with the Visual Studio Code editor, but it has since been adopted by other editors and IDEs. The LSP is under open-source license and is available on GitHub[7].

2.1.1 JSON-RPC

The LSP uses JSON-RPC to communicate between a language server and an editor. JSON-RPC (v2)[8] is a stateless, light-weight remote procedure call (RPC) [4] protocol that uses JSON as the data format.

RPC is a protocol that allows a client to call a procedure on a remote server. The client sends a request to the server, and the server sends a response back to the client. The JSON-RPC protocol defines a set of messages that are used to communicate between the client and the server. These messages include requests, responses, and notifications. The JSON-RPC protocol is designed to be simple and easy to implement, making it well-suited for use in web applications and other distributed systems.

JSON-RPC is a JSON based implementation of the RPC protocol. It defines a set of rules for encoding and decoding JSON data, as well as a set of rules for *Request*, *Notification*, and *Response* messages. The messages are sent over a transport layer, such as HTTP or WebSockets. The JSON-RPC protocol is designed to be simple and easy to implement, making it well-suited for use in web applications and other distributed systems.

All messages refer to a *method* that is a string containing the name of the method to be called. The *params* field is an array or object containing the parameters to be passed to the method. Typically, messages are synchronous, meaning that the client waits for a response from the server before continuing. The *id* field is a unique identifier for the message, which is used to match requests with responses. However, the JSON-RPC protocol also supports asynchronous messages, known as notifications, which do not

[6] https://microsoft.github.io/language-server-protocol/implementors/servers
[7] https://github.com/microsoft/language-server-protocol
[8] https://www.jsonrpc.org/specification

require a response from the server. This is implemented by setting the *id* field to *null*, in which case the server does not send a response back to the client.

The JSON-RPC specification includes the ability for clients to batch multiple requests or notifications by sending them as a list. The server is expected to respond with a corresponding list of results for each request. Additionally, the server has the flexibility to process these requests concurrently.

2.1.2 Command Specifications

The LSP is defined on top of the JSON-RPC protocol described in section 2.1.1. In abstract terms, the LSP defines a set of command that can be sent between a client and a server. In the Language Server Protocol Specification[9], these commands are divided into four categories: *Language Features*, *Text Document Synchronization*, *Workspace Features*, and *Window Features*.

Language Features

The *Language Features* provide the smarts of the language server. Usually, the client sends a request to the server to get information about the document as a tuple of `TextDocument` and `Position`. *Code comprehension* and *Coding Features* are the two main categories of commands in this category.

Here a brief description of the most important commands in this category:

- `textDocument/completion`: The completion request is sent from the client to the server to compute completion items at a given cursor position. Completion items are presented in the editor's user interface. If computing full completion items is expensive, servers can additionally provide a handler for the completion item resolve request (`completionItem/resolve`). This request is sent when a completion item is selected in the user interface.
- `textDocument/hover`: The hover request is sent from the client to the server to request hover information at a given text document position. Hover information typically includes the symbol's signature and documentation.
- `textDocument/definition`: The definition request is sent from the client to the server to resolve the definition location of a symbol at a given text document position.
- `textDocument/references`: The references request is sent from the client to the server to resolve project-wide references for the symbol denoted by the given text document position.
- `textDocument/documentHighlight`: The document highlight request is sent from the client to the server to resolve a document highlights for a given text document position. For programming languages, this usually highlights all references to the symbol denoted by the given text document position.

[9] https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification

Text Document Synchronization

The *Text Document Synchronization* commands are used to notify the server of changes to the document. Client support for `textDocument/didOpen`, `textDocument/didChange`, `textDocument/didClose` is mandatory. This includes the ability to fully and incrementally synchronize changes to the document, such as inserting, deleting, and replacing text.

Here a brief description of the most important commands in this category:

- `textDocument/didOpen`: The document open notification signals that the client is now managing a text document. The server should not read the document using its URI. Open notifications are balanced with close notifications, with only one open notification allowed at a time for a document. The server's ability to fulfill requests is unaffected by a document's open or closed status.
- `textDocument/didChange`: The document change notification is sent from the client to the server to signal changes to a text document. In response, the server should compute a new version of the document's content. The server should not rely on the client to send a specific sequence of change events. The server is free to compute the new version of the document on the fly.
- `textDocument/didClose`: The document close notification is sent from the client to the server when the document is no longer managed by the client. The document's URI is no longer valid and the server should not resolve the document using the URI.
- `textDocument/didSave`: The document save notification is sent from the client to the server when the document is saved. The notification is sent after the document has been saved.

Workspace Features

The `Workspace Features` category includes commands that allow the client to interact with the workspace. The workspace is the collection of open documents and the client's configuration. The workspace commands are used to manage the workspace, such as symbol search, workspace configuration and client configuration.

Here a brief description of the most important commands in this category:

- `workspace/symbol`: The workspace symbol request is sent from the client to the server to list project-wide symbols matching the query string. The request can be used to populate a list of symbols matching the query string in the user interface.
- `workspace/configuration`: The workspace configuration request is sent from the client to the server to fetch configuration settings from the server. The request can fetch configuration settings from the client's workspace or from the server's configuration.
- `workspace/didChangeConfiguration`: The configuration change notification is sent from the client to the server to signal changes to the client's configuration settings. The server should use the new configuration settings to update its

behavior.
- `workspace/didChangeWorkspaceFolders`: The workspace folder change notification is sent from the client to the server to signal changes to the workspace's folders. The server should use the new workspace folders to update its behavior.

Window Features

The `Window Features` category includes commands that allow the client to interact with the window. The window is the client's user interface, such as the editor, the sidebar, and the status bar. The window commands are used to manage the window, such as showing messages, showing notifications, and showing progress.

Here a brief description of the most important commands in this category:
- `window/showMessage`: The show message notification is sent from the server to the client to show a message to the user. The message can be shown in a variety of ways, such as a dialog box, a status bar, or a notification.
- `window/showMessageRequest`: The show message request is sent from the server to the client to show a message to the user and get a response. The message can be shown in a variety of ways, such as a dialog box, a status bar, or a notification.
- `window/logMessage`: The log message notification is sent from the server to the client to log a message. The message can be logged in a variety of ways, such as a console, a file, or a database.
- `window/showMessageRequest`: The show message request is sent from the server to the client to show a message to the user and get a response. The message can be shown in a variety of ways, such as a dialog box, a status bar, or a notification.

2.1.3 Key Methods Overview

Six *key methods* have been identified by langserver organization[10] as the most important methods to be implemented by a language server. These capabilities are:

1. **Diagnostic Analyze** source code — Parse and Type-check the source code to provide diagnostics.
2. **Workspace/Document Symbols** — List all symbols in the workspace.
3. **Jump to Definition** — Find and jump to the definition of a symbol.
4. **Find References** — List all usages of a symbol.
5. **Hover Information** — Show information about a symbol at the cursor.
6. **Code Completion** — Provide completions for a symbol at the cursor.

Diagnostic Analysis

The diagnostic analysis is the process of parsing and type-checking the source code to provide diagnostics. The diagnostics are used to identify errors and warnings in the

[10] https://langserver.org

Figure 2.1. *Showing diagnostics in Neovim generated by the Rust Language Server.*

source code, such as syntax errors, type errors, and unused variables. The diagnostics are presented to the user in the editor's user interface, such as a list of errors and warnings in the sidebar.

File update and diagnostic analysis are passive. Thay are sent as notifications in order to avoid blocking communication between the client and the server.

In the figure 2.1 we can see an example of diagnostics in Neovim generated by the Rust Language Server. It shows a list of errors and suggestions coused by the concatenation of two `&str` variables.

Workspace/Document Symbols

RPC Method: textDocument/workspaceSymbol or textDocument/documentSymbol

This feature is defined as both a *Language Feature* and *Workspace Feature*.

The `textDocument/workspaceSymbol` request is sent from the client to the server to list project-wide symbols matching the query string. The request can be used to populate a list of symbols matching the query string in the user interface. The granularity of the listed symbols depends on the language server implementation.

The difference between `textDocument/workspaceSymbol` and `textDocument/documentSymbol` is that the first one takes into account the visibility of the symbols in the workspace showing only the public ones. The second one shows all symbols in the document.

Jump to Definition

RPC Method: textDocument/definition

The code navigation is an important feature of the LSP.

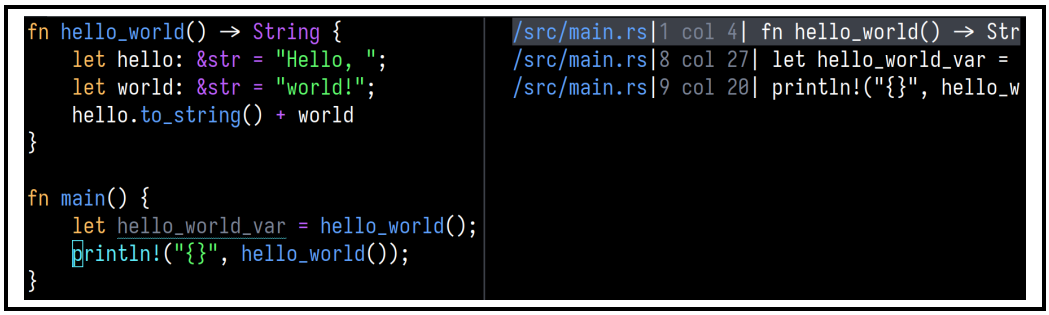


Figure 2.2. *Finding references in Neovim generated by the Rust Language Server.*

The `textDocument/definition` request is sent from the client to the server to resolve the definition location of a symbol at a given text document position. The server should return the location of the symbol's definition, such as the file path and line number.

In the figure 2.3 we can see that at the top of floating window there is the signature of the function `hello_world` meaning that the goind to definition will take us to the function definition.

Find References

RPC Method: textDocument/references

The retrieval of references is the inverse of the jump to definition explained in the previous section (2.1.3).

The `textDocument/references` request is sent from the client to the server to resolve project-wide references for the symbol denoted by the given text document position. The server should return a list of references to the symbol, such as the file path, line number, and the column number.

In the figure 2.2 we can see that the function `hello_world` is being referenced twice in the `main` function and another occurence is the function definition. In fact, three references are shown in the right window.

Hover Information

RPC Method: textDocument/hover

The hover information is a feature that shows information about a symbol at the cursor. The information typically includes the symbol's signature and documentation. This is triggered by the user hovering the mouse over a symbol in the editor or by pressing a keybinding.

A request is sent from the client to the server to request hover information at a given text document position. The server should return hover information for the symbol at

Figure 2.3. *Hover information in Neovim generated by the Rust Language Server.*

the given position, such as the symbol's signature and documentation.

In the figure 2.3 we can see that the function `hello_world` is being hovered and the signature of the function is shown in the floating window. The signature is `fn hello_world() -> String` and the documentation, written in the comment above the function, is also shown.

Code Completion

RPC Method: textDocument/completion

The code completion is a feature that provides completions for a symbol at the cursor. The completions are presented in the editor's user interface. If computing full completion items is expensive, servers can additionally provide a handler for the completion item resolve request (`completionItem/resolve`). This request is sent when a completion item is selected in the user interface.

Usually, the completion is triggered by the user typing a character or pressing a keybinding in proximity to a character, such as `.`, `::`, or `->`.

In the figure 2.4 we can see that the code completion is triggered by the user typing `to_` and pressing `<C-n>` in the editor. Two float windows are shown with the completion items. The first one shows the options for the `to_` function associated with the `&str` type and the second one shows the documentation of the selected completion item.

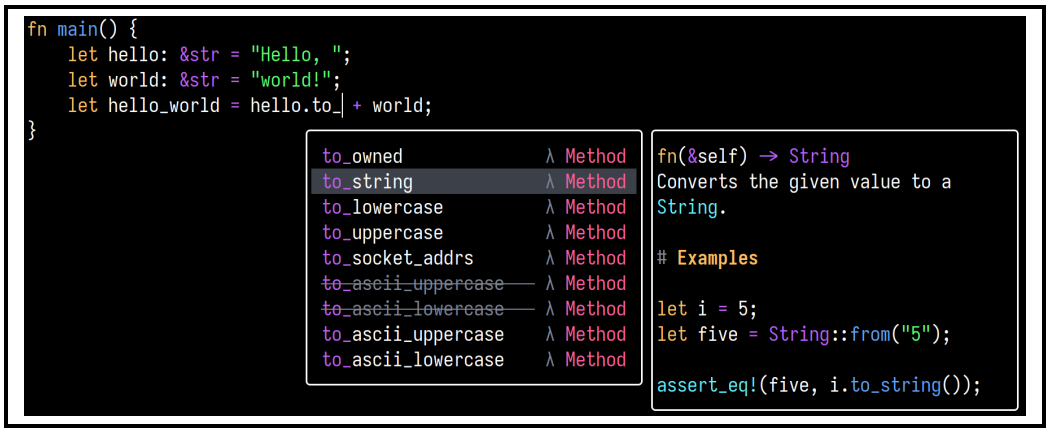


Figure 2.4. *Code completion in Neovim generated by the Rust Language Server.*

2.1.4 Approaches to Source Code Analysis

Language servers handle source code analysis using various methods, influenced significantly by the complexity of the language. The choice of approach affects how servers process file indexes, manage changes, and respond to requests.

The Language Server Protocol (LSP) supports two methods for sending updates: diffs of atomic changes and complete transmission of changed files. The former requires incremental parsing and analysis, which are challenging to implement but enable much faster processing of files upon changes. Incremental parsing relies on an internal representation of the source code that allows efficient updates for small changes. This method necessitates providing the parser with the right context to correctly parse a changed fragment of code.

In practice, most language servers re-index the entire file when changes occur, discarding the previous internal state. This approach is more straightforward to implement as it poses fewer requirements for the architects of the language server, but it is significantly less performant. Unlike incremental processing, which updates only the affected portions of its internal structure, even minor changes such as adding or removing lines require reprocessing the entire file. This method may suffice for small languages and codebases but becomes a performance bottleneck with larger ones.

For code analysis, LSP implementers must choose between lazy and greedy approaches to processing files and answering requests. Greedy implementations resolve most available information during the file's initial indexing, enabling the server to answer requests using simple lookups. Conversely, lazy approaches resolve only minimal local information during indexing, invoking ad-hoc resolution for requests and possibly memoizing the results for future use. Lazy resolution is more common with incremental indexing, as it reduces the work associated with file changes, which is crucial for complex languages that would otherwise require a significant amount of redundant work.

2.2 Language Workbenches

The term *Language Workbench* was coined by Martin Fowler in 2005 [15] to describe a set of tools that support the development of domain-specific languages (DSLs) in the context of *language-oriented programming* paradigm [34].

A language workbench will typically include tools to support the definition, reuse and composition of domain-specific languages together with their integrated development environment (IDE).

Current language workbenches usually support:

- Specification of the language concepts or metamodel
- Specification of the editing environments for the domain-specific language
- Specification of the execution semantics, e.g. through interpretation and code generation

Language workbenches have been developed for various technological spaces, but they all share the same focus developing programming languages and having a strong emphasis on code resuability and separation of concerns.

Some of the most popular language workbenches are:

- **JetBrains MPS**[11]: A powerful language workbench that allows the creation of domain-specific languages and the generation of code from them.
- **Xtext**[12]: A framework for developing domain-specific languages and languages for code generation.
- **Spoofax**[13]: A language workbench for developing textual domain-specific languages.
- **MontiCore**[14]: A language workbench for developing domain-specific languages and language families.

`Neverlang` (Sect. 2.2.1) will be discussed with particular detail, since most of the research discussed in this dissertation will use `Neverlang` as a running example.

2.2.1 Neverlang

The *Neverlang* [8, 10, 29] framework promote the code reusability and the separation of concerns in the implementation of programming languages, based around the concept of *language-feature*. The basic development unit is the **module**, as shown in line 1 of Listing 2.2. A module may contain a **reference syntax** and could have zero or multiple **role**s. A role, used to define the semantics, is unit of composition that defines actions that should be executed when some syntax is recognized, as defined by *syntax-directed translation* [1]. Syntax definitions are defined using *Backus–Naur form* (BNF) grammars, represented as sets of *productions* and *terminals*. Syntax definitions

[11] https://www.jetbrains.com/mps/
[12] https://www.eclipse.org/Xtext/
[13] https://www.metaborg.org/en/latest/
[14] https://www.monticore.de/

```
1  module Backup {
2    reference syntax {
3      Backup: Backup ⇐ "backup" String String;
4      Cmd: Cmd      ⇐ Backup;
5      categories    : Keyword = { "backup" };
6    }
7    role(execution) {
8      0 .{
9          String src  = $1.string, dest = $2.string;
10         $$FileOp.backup(src, dest);
11     }.
12   }
13 }
14 slice BackupSlice {
15   concrete syntax from Backup
16   module Backup with role execution
17   module BackupPermCheck with role permissions
18 }
19 language LogLang {
20   slices BackupSlice RemoveSlice RenameSlice
21          MergeSlice Task Main LogLangTypes
22   endemic slices FileOpEndemic PermEndemic
23   roles
24     syntax < terminal-evaluation < permissions : execution
25 }
```

Listing 2.2. *Syntax and semantics for the backup task. Borrowed from [14].*

and semantic **role**s are tied together using *slice*s. Listing 2.2 shows a simple example of a Neverlang module implementing a backup task of the `LogLang` LPL. Reference syntax is defined in lines 2-6; the *categories* (line 5) are used also to generate the syntax highlighting for the IDEs. Semantic actions may be attached to a non-terminal using the production's name as a reference, or using the position of the non-terminal in the grammar, as shown in line 8, numbering start with 0 from the top left to the bottom right. The two *String* non-terminals on the right-hand side of the *Backup* production are referenced using 1 and 2, respectively. Different semantic actions may be attached to the same production thanks to the multiplicity of **role**s that can be defined for a module. Each **role** is a compilation phase that can be executed in a specific order, as shown in line 24. In contrast, the `BackupSlice` (lines 14-18) reveals how the syntax and semantics are tied together; choosing the `concrete syntax` from the Backup module (line 15), and two `roles` from two different modules (lines 16-17). Finally, the `language` can be created by composing multiple **slices** (line 20). The composition in Neverlang is twofold [14]: between modules and between slices. Thus, the grammars are merged in order to generate the complete language parser. On the other hand, the semantic actions are composed in a pipeline, and each **role** traverses the syntax tree in the order specified in the **roles** clause (line 24). Plase see [29] for a more detailed explanation of the `Neverlang` framework.

Language	Type System	Type Checking	Type Inference
C	Weak	Static	No
OCaml	Strong	Static	Yes
Java	Strong	Static	Yes
Rust	Strong	Static	Yes
Python	Strong	Dynamic	Yes
JavaScript	Weak	Dynamic	Yes
Haskell	Strong	Static	Yes
Erlang	Strong	Dynamic	Yes
Perl	Weak	Dynamic	No

Table 2.1. *Examples of programming languages and their type systems.*

2.3 Static Analysis and Type Systems

Static code analysis is a software verification technique refers to the process of examining the code without executing it in order to capture the defects in the code early avoiding costly later fixations [2]. Static code analysis has two main approaches: manual and automated [17]. Manual code analysis is a time-consuming process that requires human expertise to review the code and identify defects. Automated code analysis, on the other hand, uses tools to analyze the code and identify defects automatically. Automated code analysis tools can be used to analyze the code for defects such as syntax errors, type errors, and logic errors. These tools can also be used to enforce coding standards and best practices, such as naming conventions, code formatting, and code complexity.

2.3.1 Type Systems

Type systems are a fundamental part of static code analysis. A type system [25] is a set of rules that define how types are used in a programming language. The type system defines the types of variables, functions, and expressions, as well as the rules for type compatibility and type inference.

Type checking is the process of verifying that the types of variables, functions, and expressions are used correctly in the code. Type checking can be done *statically*, at compile time, or *dynamically*, at runtime. Static type checking is more efficient and catches type errors earlier in the development process, while dynamic type checking is more flexible and allows for more dynamic programming.

Type inference is the process of automatically deducing the types of variables, functions, and expressions in the code. Type inference is used to reduce the amount of type annotations required in the code, making the code more concise and easier to read. Type inference is used in statically typed languages such as OCaml, Java, and Rust, as well as in dynamically typed languages such as Python and JavaScript, see table 8.2.

A programming language can be classified based on its type system and type

checking. There are two main categories of type systems: *strong* and *weak*. A strong type system enforces type safety and does not allow for implicit type conversions, while a weak type system allows for implicit type conversions and does not enforce type safety. A programming language can also be classified based on its type checking: *static* or *dynamic*. Static type checking is done at compile time and catches type errors before the code is executed, while dynamic type checking is done at runtime and catches type errors as the code is executed.

In table 8.2 we show examples of programming languages and their type systems. C is a statically typed language with a weak type system, meaning that it allows for implicit type conversions and does not enforce type safety. OCaml, Java, and Rust are statically typed languages with strong type systems, meaning that they enforce type safety and do not allow for implicit type conversions. Python and JavaScript are dynamically typed languages with strong type systems, meaning that they enforce type safety at runtime. Haskell is a statically typed language with a strong type system and support for type inference. Erlang is a dynamically typed language with a strong type system and support for type inference.

2.3.2 Theoretical Aspects

Theoretical aspects of type systems are an important research area in computer science. Type systems are used to ensure the correctness and safety of programs by enforcing rules about how types are used in the code. Type systems can be classified based on their properties, such as *soundness*, *completeness*, and *decidability*.

A brief explanation of these properties is given below:

- A type system is **sound** if it guarantees that well-typed programs do not produce runtime errors. Soundness is an important property of a type system because it ensures that the type system is correct and that it enforces the correct use of types in the code.
- A type system is **complete** if it can infer the type of any expression in the code. Completeness is an important property of a type system because it ensures that the type system can handle all possible expressions in the code.
- A type system is **decidable** if it can determine the type of any expression in a finite amount of time. Decidability is an important property of a type system because it ensures that the type system is efficient and can be used in practice.

2.3.3 Type Theory as a Logic

A *Type theory* is a mathematical logic, which is to say it is a collection of rules of inference that result in judgments. Most logics are based on judgements, which are statements that assert the truth of a proposition. Martin-Löf in *Intuitionistic Type Theory* [23] introduced a new type theory, previously published in 1972, that is based on the idea of judgments as the central concept of the theory. In Martin-Löf's [22] type system, a judgment is no longer just an affirmation or denial of a proposition, but a general act of

knowledge. When reasoning mathematically we make judgments about mathematical objects. One form of judgment is to state that some mathematical statement is true. Another form of judgment is to state that something is a mathematical object, for example a set [12].

In the following sections, we will discuss the basic concepts of type theory and how it is used in the context of programming languages.

Judgements

Martin-Löf type theory has four basic forms of judgments and is a considerably more complicated system than first-order logic.

The four basic forms of judgments are:

- $\vdash A$ type — This judgment asserts that A is a well-formed type.
- $\vdash a : A$ — This judgment asserts that a is an element of type A.
- $\vdash A = B$ type — This judgment asserts that A and B are equal types.
- $\vdash a = b : A$ — This judgment asserts that a and b are equal elements of type A.

In programming languages, the first two judgments are used to define the types of variables and functions, while the last two judgments are used to define equality between types and elements. For instance,

$$\vdash \texttt{Int}\ \text{type} \quad \vdash 1 : \texttt{Int} \quad \vdash \texttt{Int} = \texttt{Int}\ \text{type} \quad \vdash 1 = 1 : \texttt{Int}$$

Contexts

In general, judgments are *hypothetical*, meaning that they are made under certain assumptions. These assumptions are kept track of in a *context*. In Martin-Löf's type theory, the context is a list $x_1 : A_1, \ldots, x_n : A_n$ of variables and their types. Note that the context is ordered, meaning that the order of the variables matters.

The four basic forms of judgments are made in the context of a given context Γ:

- $\Gamma \vdash A$ type — This judgment asserts that A is a well-formed type in the context Γ.
- $\Gamma \vdash a : A$ — This judgment asserts that a is an element of type A in the context Γ.
- $\Gamma \vdash A = B$ type — This judgment asserts that A and B are equal types in the context Γ.
- $\Gamma \vdash a = b : A$ — This judgment asserts that a and b are equal elements of type A in the context Γ.

In programming languages, the context is used to keep track of the types of variables and functions in the code. For instance,

$$x : \texttt{Int}, y : \texttt{Int} \vdash x + y : \texttt{Int}$$

or,

$$x : \texttt{Int}, y : \texttt{Int} \vdash x + y = y + x : \texttt{Int}$$

Inference Rules

Let Π be a set of type functions in the form of $\Pi x : A.B$ where x is a variable of type A and B is a type.

The first type of inference rule in Martin-Löf's type theory is the *formation rule*.

Formation Rule

This rules define how types are formed and how elements are formed in a given context. The formation rule is used to define the types of variables and functions in the code.

$$\frac{\Gamma \vdash A \text{ type} \qquad \Gamma, x : A \vdash B \text{ type}}{\Gamma \vdash \Pi x : A.B \text{ type}}$$

Written in natural language, the Π-formation rule states that if A is a type in the context Γ and B is a type in the context $\Gamma, x : A$, then $\Pi x : A.B$ is a type in the context Γ.

The second type of inference rule in Martin-Löf's type theory is the *introduction rule*.

Introduction Rule

This rule define how elements are introduced in a given context. The introduction rule is used to define the elements of variables and functions in the code.

$$\frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda x : A.b : \Pi x : A.B}$$

Written in natural language, the Π-introduction rule states that if b is an element of type B in the context $\Gamma, x : A$, then $\lambda x : A.b$ is an element of type $\Pi x : A.B$ in the context Γ.

The third type of inference rule in Martin-Löf's type theory are the *elimination rule*.

Elimination Rule

This rule define how elements are eliminated in a given context. The elimination rule is used to define the elimination of variables and functions in the code.

$$\frac{\Gamma \vdash f : \Pi x : A.B \qquad \Gamma \vdash a : A}{\Gamma \vdash fa : B[x := a]}$$

Written in natural language, the Π-elimination rule states that if f is an element of type $\Pi x : A.B$ in the context Γ and a is an element of type A in the context Γ, then fa is an element of type $B[x := a]$ in the context Γ.

```rust
fn main() {
    let x: i32 = 3;
    let y: i32 = 7;
    let z = x + y;
}
```

```
let x : int = 3;;
let y : int = 7;;
let z = x + y;;
```

Listing 2.3. *Example of type inference in Rust and OCaml.*

The fourth type of inference rule in Martin-Löf's type theory are the *equality rule*.

Equality Rule

This rule define how types and elements are equal in a given context. The equality rules are used to define the equality of types and elements in the code. In the following $B[x := a]$ denotes the substitution of a for x in B.

$$\frac{\Gamma, x : A \vdash b : B \qquad \Gamma \vdash a : A}{\Gamma \vdash (\lambda x : A.b)a = b[x := a] : B[x := a]}$$

Written in natural language, the Π-equality rule states that if b is an element of type B in the context $\Gamma, x : A$ and a is an element of type A in the context Γ, then $(\lambda x : A.b)a$ is equal to $b[x := a]$ in the context Γ.

2.3.4 Type Inference in Programming Languages

Given the typing environment Γ, the type inference algorithm uses the rules of type theory to deduce the types of variables, functions, and expressions in the code. The type inference algorithm is used to determine the types of variables and functions in the code, as well as to check the correctness of the code.

In listing 2.3 we show an example of type inference in Rust and OCaml. In both examples, the type inference algorithm is used to deduce the types of the variables `z` in the code. In case of Rust code (line 5), the type of `z` is inferred to be `i32` because the types of `x` and `y` are known to be `i32`. In case of OCaml code (line 3), the type of `z` is inferred to be `int` because the types of `x` and `y` are known to be `int`.

2.4 Software and Language Product Lines

3 Related Work

According to Tomasetti [28], an external Domain-Specific Language in contrast to an internal one has tool support. So, in order to build this support language workbenches come to the rescue. Nowadays, developers are increasingly using IDEs to write code, both for general-purpose languages (GPLs) and for domain-specific languages (DSLs), as they provide features such as syntax highlighting, code completion, and code navigation, thanks also to the Language Server Protocol (LSP) [26]. To develop this kind of tool support, there are various language workbenches available that simplify the creation of abstract syntax, parsers, editors, and generators [15].

In this chapter, we will present some of the most relevant language workbenches and tools that are available to support the language development process (Section 3.1). We will also present some of the most relevant tools that support modularization and composition of languages (Section 3.2). Finally, we will present some of the most relevant tools that support IDE and LSP generation (Section 3.3).

3.1 Syntax and Semantics Definition in Language Workbenches

3.1.1 JustAdd

JustAdd [13] is a modular compiler construction system developed at the Computer Science department of the Lund University. It doesn't offer built-in support for concrete syntax definition or parser generation, however it can be integrated with third-party parser generators; language development in JastAdd, instead, is focused on modeling the AST as an object-oriented class hierarchy. Semantic phases are defined as methods in the AST classes, and the compiler is generated by the JastAdd compiler generator.

3.1.2 Melange

Melange [11] is a language workbench developed by the DiverSE research team at the Institut National de Recherche en Informatique et en Automatique (INRIA). Based on the Eclipse Modeling Framework (EMF) [27], Melange is meant to be used for the development of domain-specific languages. Abstract syntax support is provided by EMF, while concrete syntax is defined using the Xtext (Sect. 3.1.7) language workbench. Semantics is specified using the Kermeta 3 (K3) language [18] which is based on Xtend[1].

[1] https://www.eclipse.org/xtend/

3.1.3 MontiCore

MontiCore [21] is a language workbench developed by the Software Engineering group at the RWTH Aachen University. Using a single DLS it is possible to define the abstract syntax, the concrete syntax; it takes one or more grammars as input and generates Java source code. A visitor pattern is written in Java to define the semantics of the language.

3.1.4 MPS – Meta Programming System

Meta Programming System [31, 32] is a developement environment developed by JetBrains[2]. It is a *projectional editor* [33] that allows the definition of abstract syntax and concrete syntax through a graphical editor. In order to add semantics, MPS provides a set of methods written in a Java-like language called *BaseLanguage*.

3.1.5 Rascal

Rescal [20] is a meta-programming language developed at the Centrum Wiskunde & Informatica (CWI) in Amsterdam. The abstract syntax is defined using algebraic data types (ADTs), while the concrete syntax is defined SDF [16] (also used in Spoofax, see Sect. 3.1.6). The semantics phase is defines as a function `eval` that takes an AST as input and with pattern matching evaluates the AST.

3.1.6 Spoofax

Spoofax [19] is a language workbench developed at the Delft University of Technology. It is an Eclipse based solution that provides support that use *Syntax Definition Formalism 3* (SDF3) [16] for defining grammars. In Spoofax, the semantics is defined using the *Stratego* [30] language, which is a term rewriting language.Spoofax, the sew

3.1.7 Xtext

Xtext [3] is a language workbench developed by the Eclipse Foundation. The grammar is written in Xtext language, which is capable of generating both the ANTLR-based [24] parser and EMF [27] model used to represent the AST. Semantics is in fact defined by implementing a code generator; this integrates *out of the box* with Eclipse's built-in system.system. Xtext is distributed with *Xtend* [3], a Java-like language that can be used to define the semantics of the language.

3.2 Modularization and Composition in Language Workbenches

Finding and working with the right abstractions for describing a problem or its solution is one of the central pillars of software engineering [32]. In fact, modularization and composition are important and advanced features in language workbenches. They

[2] https://www.jetbrains.com

allow the developer to define a language by composing multiple features, each of which can be developed independently and then combined to create a new language.

Developers typically tackle the complexity of building an interpreter by using a vertical, functional decomposition approach. This method generally involves creating distinct phases for a lexer, parser, semantic analyzer, and code evaluator. While organizing the process into these phases is beneficial, it is not sufficient on its own [9]. According to Bosch [5], a vertical decomposition still results in complex compilers or interpreters that are hard to maintain, extend and whose pieces are hardly reusable. He suggests that a vertical, functional decomposition should be accompanied by a horizontal, structural decomposition.

A component can be any language feature (see Sect. 2.4) that can be reused across different languages. Syntax and semantics are commonly used together to define a language feature, but other features can be defined as well. Each feature can be precompiled [10] or not. There are several language workbenches that support modularization and composition of languages, either with precompiled components or not. Different language workbenches provide different levels of support for modularization and composition of languages. In the following, we will present some of the most relevant language workbenches.

Following the order of the previous section, we will present the language workbenches that support modularization and composition of languages. **JustAdd** allows the developer to define a language by composing multiple features, but it requires that syntax and semantics have the same signature. On the other hand, thanks to an Encore model, that is a declarative specification of a set of classes, each of which can contain fields, methods, and constraints, **Melange** support composition. Encore models can be composed by renaming elements, which allows the developer to define a new language by composing multiple features. Another language workbench that supports modularization and composition of languages is **MontiCore**. It supports multiple grammars inheritance, but it does not give support for adding new attributes to imported symbols. Due to the fact that **MPS** is a projectional editor, without parsing phase, it allows arbitrary notations to be combined. It supports composition of languages, but it does not support precompiled components. Moving to **Rascal**, rewrite rules is the principal mechanism to compose features; the obstacle is that certain identifiers, such as constructors for the operations ADT, must be consistent across seemingly unrelated modules. **Spoofax** is unique in its support for modularization and composition of languages. It uses *Stratego* [6] to rewrite the tree, allowing the conversion from an AST to another. **Xtext**, instead, supporting only single grammar inheritance, does not support composition of languages.

The only language workbenches that support precompiled components are **Neverlang** [10]. While in **MontiCore** and **Xtext** code implementing semantics can be excluded from package, but the grammar cannot be excluded, so it is possible have a partial support for precompiled components.

In Table 3.2 we summarize the support for modularization and composition of the language workbenches presented in this section. In the first column, we list the language workbenches. Using ●, ◐, and ○, in the second and third columns, we

Language Workbench	Modularization Supp.	Precompiled Features Supp.
JustAdd	◐	○
Melange	●	○
MontiCore	◐	◐
MPS	●	○
Rascal	○	○
Spoofax	●	◐
Xtext	○	◐
Neverlang	●	●

Table 3.1. *Language Workbenches Supporting Modularization, Composition and Precompiled Features.*

Language Workbench	Native IDE gen.	Native LSP Gen.	LSP Mod.
JustAdd	○	○	
Melange	●	○	
MontiCore	3	○	
MPS	●	○	
Rascal	●	○	
Spoofax	●	○	
Xtext	●	●	

Table 3.2. *Language Workbenches Supporting Modularization and Composition.*

indicate whether the language workbench supports, partially supports, or does not support modularization and composition, respectively.

3.3 IDE and LSP Support

Xtext is by now the only language workbench supporting the LSP. [7]

4
Concept

5
Implementation

6
Evaluation

7
Conclusions

8 Tests

Test

This is a test.

- *Test1*—This is a test.
- *Test2*—This is another test.
- *Test3*—This is yet another test.

My Heading

This is a **tcolorbox**.

Here, you see the lower part of the box.

My Heading

This is a **tcolorbox**.

Here, you see the lower part of the box.

My Heading

This is a **tcolorbox**.

Here, you see the lower part of the box.

My Heading

This is a **tcolorbox**.

Here, you see the lower part of the box.

1 **Example**

hello

```rust
fn main() {
    let x = 42;
    let y = 3.14;
    let z = x + y;
    println!("The sum is: {}", z);
}
```

Listing 8.1. *Example of type inference in Rust and OCaml.*

Language	Type System	Type Checking	Type Inference
C	Weak	Static	No
OCaml	Strong	Static	Yes
Java	Strong	Static	Yes
Rust	Strong	Static	Yes
Python	Strong	Dynamic	Yes
JavaScript	Weak	Dynamic	Yes
Haskell	Strong	Static	Yes
Erlang	Strong	Dynamic	Yes
Perl	Weak	Dynamic	No

Table 8.1. *Examples of programming languages and their type systems.*

2 Solution

hello

3 Example

hello

4 Solution

hello

This is a **tcolorbox**.

Language	Type System	Type Checking	Type Inference
C	Weak	Static	No
OCaml	Strong	Static	Yes
Java	Strong	Static	Yes
Rust	Strong	Static	Yes
Python	Strong	Dynamic	Yes
JavaScript	Weak	Dynamic	Yes
Haskell	Strong	Static	Yes
Erlang	Strong	Dynamic	Yes
Perl	Weak	Dynamic	No

Table 8.2. *Examples of programming languages and their type systems.*

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, Reading, Massachusetts, 1986.

[2] Aybuke Aurum, Håkan Petersson, and Claes Wohlin. State-of-the-art: software inspections after 25 years. *Software Testing, Verification and Reliability*, 12(3):133–154, 2002.

[3] Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. PACKT Publishing Ltd, August 2013.

[4] Andrew D. Birrell and Bruce Jay Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[5] Jan Bosch. Delegating Compiler Objects. In Tibor Gyimóth, editor, *Proceedings of the 6th International Conference on Compiler Construction (CC'96)*, Lecture Notes in Computer Science 1060, pages 326–340, Linköping, Sweden, April 1996. Springer.

[6] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A Language and Toolset for Program Transformation. *Journal Science of Computer Programming*, 72(1-2):52–70, June 2008.

[7] Hendrik Bünder. Decoupling language and editor-the impact of the language server protocol on textual domain-specific languages. 2019.

[8] Walter Cazzola. Domain-Specific Languages in Few Steps: The Neverlang Approach. In Thomas Gschwind, Flavio De Paoli, Volker Gruhn, and Matthias Book, editors, *Proceedings of the 11th International Conference on Software Composition (SC'12)*, Lecture Notes in Computer Science 7306, pages 162–177, Prague, Czech Republic, 31st of May-1st of June 2012. Springer.

[9] Walter Cazzola and Albert Shaqiri. Modularity and Optimization in Synergy. In Don Batory, editor, *Proceedings of the 15th International Conference on Modularity (Modularity'16)*, pages 70–81, Málaga, Spain, 14th-17th of March 2016. ACM.

[10] Walter Cazzola and Edoardo Vacchi. Neverlang 2: Componentised Language Development for the JVM. In Walter Binder, Eric Bodden, and Welf Löwe, editors, *Proceedings of the 12th International Conference on Software Composition (SC'13)*, Lecture Notes in Computer Science 8088, pages 17–32, Budapest, Hungary, 19th of June 2013. Springer.

[11] Thomas Degueule, Benoît Combemale, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. Melange: a Meta-Language for Modular and Reusable Development of DSLs. In Davide Di Ruscio and Markus Völter, editors, *Proceedings of the 8th International Conference on Software Language Engineering (SLE'15)*, pages 25–36, Pittsburgh, PA, USA, October 2015. ACM.

[12] Peter Dybjer and Erik Palmgren. Intuitionistic Type Theory. In Edward N. Zalta and Uri Nodelman, editors, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Spring 2023 edition, 2023.

[13] Torbjörn Ekman and Görel Hedin. The JastAdd System — Modular Extensible Compiler Construction. *Science of Computer Programming*, 69(1-3):14–26, December 2007.

[14] Luca Favalli, Thomas Kühn, and Walter Cazzola. Neverlang and FeatureIDE Just Married: Integrated Language Product Line Development Environment. In Philippe Collet and Sarah Nadi, editors, *Proceedings of the 24th International Software Product Line Conference (SPLC'20)*, pages 285–295, Montréal, Canada, 19th-23rd of October 2020. ACM.

[15] Martin Fowler. Language Workbenches: The Killer-App for Domain Specific Languages? Martin Fowler's Blog, May 2005.

[16] Jan Heering, Paul R. H. Hendricks, Paul Klint, and Jan Rekers. The Syntax Definition Formalism SDF —Reference Manual—. *SIGPLAN Notices*, 24(11):43–75, November 1989.

[17] Bilal Ilyas and Islam Elkhalifa. *Static Code Analysis: A Systematic Literature Review and an Industrial Survey*.

[18] Jean-Marc Jézéquel, Benoît Combemale, Olivier Barais, Martin Monperrus, and François Fouquet. Mashup of Metalanguages and Its Implementation in the kermeta Language Workbench. *Software and Systems Modeling*, 14(2):905–920, May 2015.

[19] Lennart C. L. Kats and Eelco Visser. The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. In Martin Rinard, Kevin J. Sullivan, and Daniel H. Steinberg, editors, *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'10)*, pages 444–463, Reno, Nevada, USA, October 2010. ACM.

[20] Paul Klint, Tijs van der Storm, and Jurgen Vinju. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In Andrew Walenstein and Sibylle Schupp, editors, *Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM'09)*, pages 168–177, Edmonton, Canada, September 2009. IEEE.

[21] Holger Krahn, Bernhard Rumpe, and Steven Völkel. MontiCore: A Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer*, 12(5):353–372, September 2010.

[22] Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. *Nordic journal of philosophical logic*, 1(1):11–60, 1996.

[23] Per Martin-Löf and Giovanni Sambin. *Intuitionistic type theory*, volume 9. Bibliopolis Naples, 1984.

[24] Terence J. Parr and Russell W. Quong. ANTLR: A Predicated-LL(k) Parser Generator. *Software—Practice and Experience*, 25(7):789–810, July 1995.

[25] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

[26] Roberto Rodriguez-Echeverria, Javier Luis Cánovas Izquierdo, Manuel Wimmer, and Jordi Cabot. Towards a language server protocol infrastructure for graphical modeling. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, MODELS '18, page 370–380, New York, NY, USA, 2018. Association for Computing Machinery.

[27] Dave Steinberg, Dave Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley, December 2008.

[28] Federico Tomassetti. The complete guide to (external) domain specific languages.

[29] Edoardo Vacchi and Walter Cazzola. Neverlang: A Framework for Feature-Oriented Language Development. *Computer Languages, Systems & Structures*, 43(3):1–40, October 2015.

[30] Eelco Visser. Stratego: A Language for Program Transformation Based on Rewriting Strategies. In Aart Middeldorp, editor, *Proceedings of the 12th International Conference on Rewriting Techniques and Applications (RTA'01)*, Lecture Notes in Computer Science 2051, pages 357–361, Utrecht, The Netherlands, May 2001. Springer.

[31] Markus Völter. Language and IDE Modularization and Composition with MPS. In Ralf Lämmel, João Saraiva, and Joost Visser, editors, *Proceedings of the 4th International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'11)*, Lecture Notes in Computer Science 7680, pages 383–430, Braga, Portugal, July 2011. Springer.

[32] Markus Völter and Vaclav Pech. Language Modularity with the MPS Language Workbench. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*, pages 1449–1450, Zürich, Switzerland, June 2012. IEEE.

[33] Markus Völter, Janet Siegmund, Thorsten Berger, and Bernd Kolb. Towards User-Friendly Projectional Editors. In Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju, editors, *Proceedings of the 7th International Conference on Software Language Engineering (SLE'14)*, Lecture Notes in Computer Science Volume 8706, pages 41–61, Västerås, Sweden, September 2014. Springer.

[34] Martin P. Ward. Language Oriented Programming. *Software—Concept and Tools*, 15(4):147–161, October 1994.