Toward a Modular Approach for Type Systems and LSP Generation

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 2.2, static analysis and type systems in Section 2.3, and software and language product lines in Section 2.5.

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-agnostic 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`

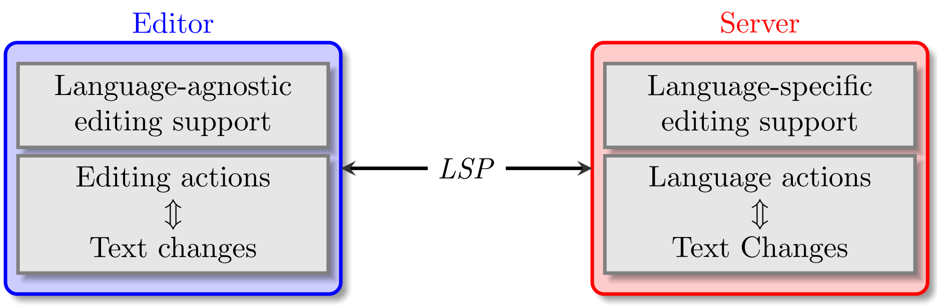

Editor
Language-agnostic editing support
Editing actions ⇕ Text changes
LSP
Server
Language-specific editing support
Language actions ⇕ Text Changes

Figure 2.1. *LSP approach to language support. Borrowed from [?].*

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].

Before the advent of LSP and DAP, developers had to implement language support for each editor separately, having the number of combinations to support **L** languages in $\mathbf{L} \times \mathbf{E}$, where **E** is the number of editors. Currently, the number of combinations to support **L** languages is $\mathbf{L} + \mathbf{E}$ [?], as the Microsoft LSP and DAP are editor-agnostic, as shown in Figure 2.1.

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) [?] 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

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

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 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.

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

- `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.

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:

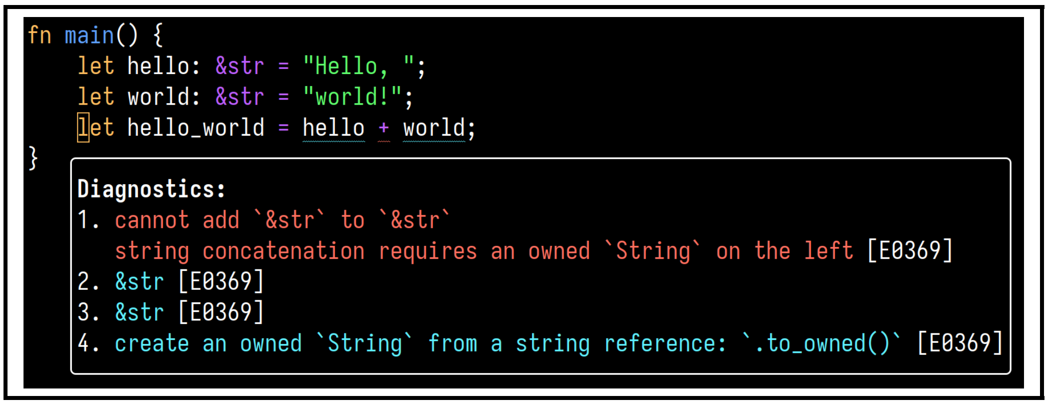
– `window/showMessage`: The show message notification is sent from the server to the client to show a message to the user. The message can be shown in a variety of ways, such as a dialog box, a status bar, or a notification.
– `window/showMessageRequest`: The show message request is sent from the server 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.

[10] https://langserver.org



```
fn main() {
    let hello: &str = "Hello, ";
    let world: &str = "world!";
    let hello_world = hello + world;
}   ┌──────────────────────────────────────────────────────────────────────────────┐
    │Diagnostics:                                                                  │
    │1. cannot add `&str` to `&str`                                                │
    │   string concatenation requires an owned `String` on the left [E0369]        │
    │2. &str [E0369]                                                               │
    │3. &str [E0369]                                                               │
    │4. create an owned `String` from a string reference: `.to_owned()` [E0369]    │
    └──────────────────────────────────────────────────────────────────────────────┘
```

Figure 2.2. *Showing diagnostics in Neovim generated by the Rust Language Server.*

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 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 2.2 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.

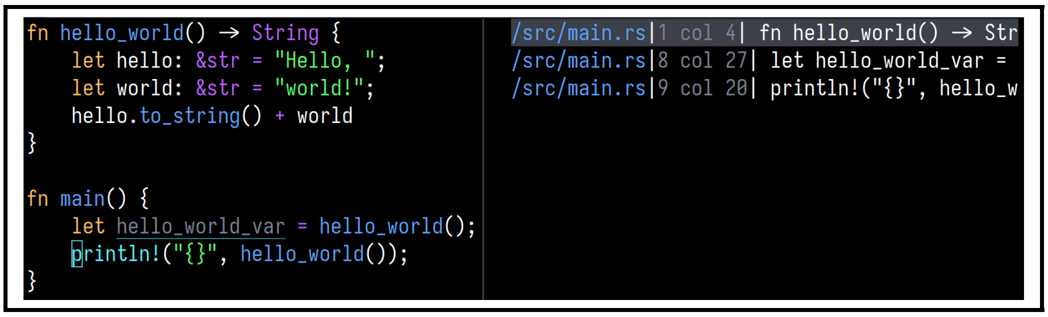
Workspace/Document Symbols

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.



```
fn hello_world() -> String {
    let hello: &str = "Hello, ";
    let world: &str = "world!";
    hello.to_string() + world
}

fn main() {
    let hello_world_var = hello_world();
    println!("{}", hello_world());
}
```

```
/src/main.rs|1 col 4| fn hello_world() -> Str
/src/main.rs|8 col 27| let hello_world_var =
/src/main.rs|9 col 20| println!("{}", hello_w
```

Figure 2.3. *Finding references in Neovim generated by the Rust Language Server.*

Jump to Definition

RPC Method: textDocument/definition

The code navigation is an important feature of the LSP.

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 2.4 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

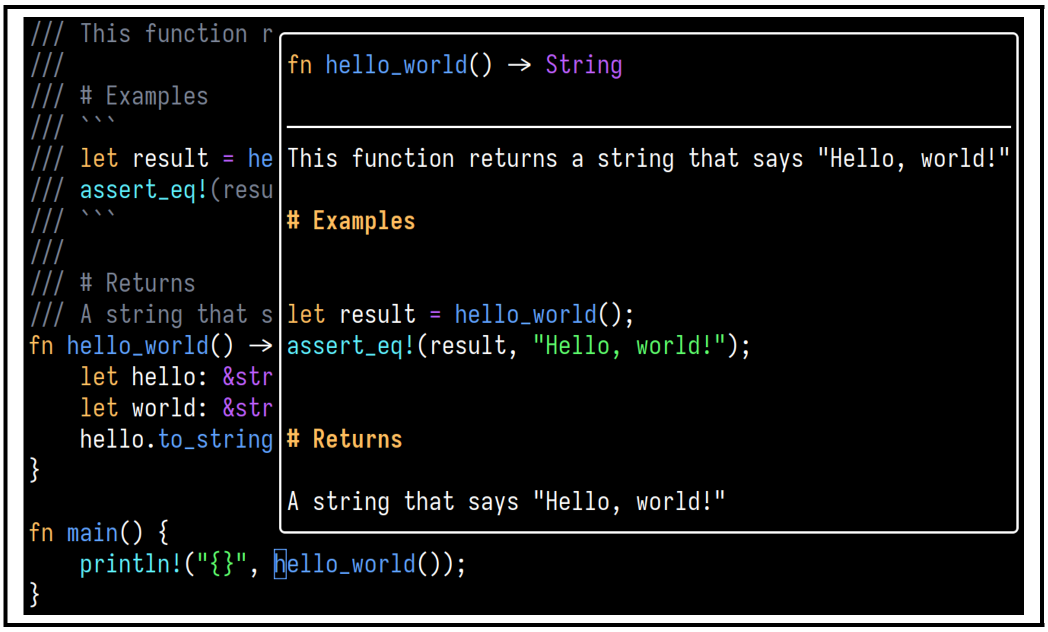
The retrieval of references is the inverse of the jump to definition explained in the previous section (2.1.3).

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 2.3 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



```
/// This function r
///                 fn hello_world() -> String
/// # Examples
/// ```
/// let result = he This function returns a string that says "Hello, world!"
/// assert_eq!(resu
/// ```             # Examples
///
/// # Returns
/// A string that s let result = hello_world();
fn hello_world() -> assert_eq!(result, "Hello, world!");
    let hello: &str
    let world: &str
    hello.to_string # Returns
}
                    A string that says "Hello, world!"
fn main() {
    println!("{}", hello_world());
}
```

Figure 2.4. *Hover information in Neovim generated by the Rust Language Server.*

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

In the figure 2.4 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.

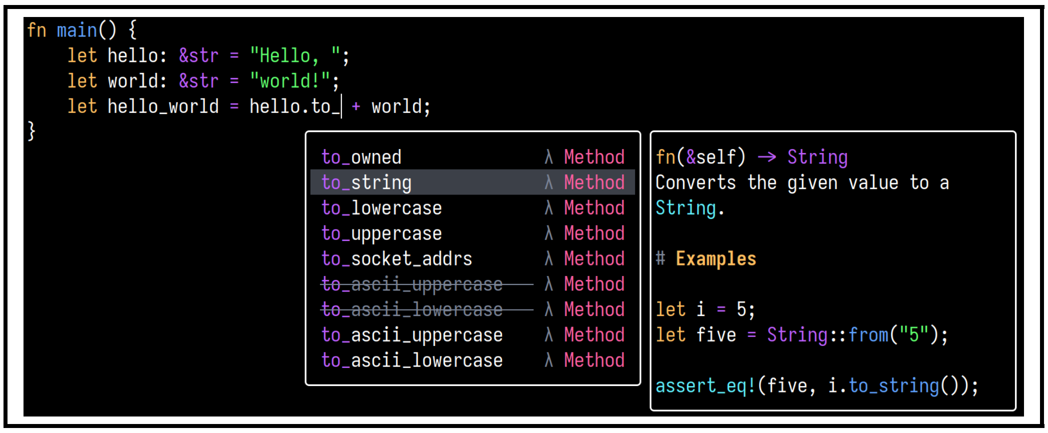
Code Completion

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 2.5 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.



```
fn main() {
    let hello: &str = "Hello, ";
    let world: &str = "world!";
    let hello_world = hello.to_ + world;
}
```

```
to_owned                λ Method
to_string               λ Method
to_lowercase            λ Method
to_uppercase            λ Method
to_socket_addrs         λ Method
~~to_ascii_uppercase~~      λ Method
~~to_ascii_lowercase~~      λ Method
to_ascii_uppercase      λ Method
to_ascii_lowercase      λ Method
```

```
fn(&self) -> String
Converts the given value to a
String.

# Examples

let i = 5;
let five = String::from("5");

assert_eq!(five, i.to_string());
```

Figure 2.5. *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 [?] to describe a set of tools that support the development of domain-specific languages (DSLs) in the context of *language-oriented programming* paradigm [?].

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* [?, ?, ?] 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

[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 2.1. *Syntax and semantics for the backup task. Borrowed from [?].*

Listing 2.1. 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* [?]. Syntax definitions are defined using *Backus–Naur form* (BNF) grammars, represented as sets of *productions* and *terminals*. Syntax definitions and semantic **role**s are tied together using *slice*s. Listing 2.1 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 `role`s 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 [?]: 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 [?] for a more detailed explanation of the `Neverlang` framework.

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 [?]. Static code analysis has two main approaches: manual and automated [?]. 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 [?] 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 2.1.

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 2.1 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

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.*

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* [?] 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 [?] 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 [?].

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 $x_1 : A_1, \dots, x_n : A_n$ 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.

```
1 fn main() {
2     let x: i32 = 3;
3     let y: i32 = 7;
4     let z = x + y;
5 }
```

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

Listing 2.2. *Example of type inference in Rust and OCaml.*

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

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.

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 2.2 we show an example of type inference in Rust and OCaml. In both examples, thesec:background:software-and-language-product-lines 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`.

```scala
class Calculator(val value: Int) {
  infix def add(x: Int): Calculator = new Calculator(value + x)
  infix def subtract(x: Int): Calculator = new Calculator(value - x)
  infix def multiply(x: Int): Calculator = new Calculator(value * x)
  infix def divide(x: Int): Calculator = new Calculator(value / x)
}
@main def CalcExample =
  val calc = Calculator(0)
  val result = calc add 5 multiply 2 subtract 1 divide 2
  println(result.value)
```

Listing 2.3. *An internal DSL for arithmetic expressions in Scala.*

2.4 Domain-Specific Languages

According to [?], many software language are *domain specific* rather than general purpose. They can be classified in:

- application oriented [?],
- special purpose [?],
- specialized [?],
- task specific [?], and
- application languages [?],

Application language are called also *fourth-generation languages* [?], and an example is SQL [?] used to query databases.

Domain specific languages (DSLs) are languages that are designed to be used in a specific domain or application area. DSLs are used to express concepts and operations that are specific to a particular domain. DSLs are typically more expressive and easier to use than general-purpose programming languages, making them well-suited for developing software with respect to the problems of a particular application domain [?].

According to [?], the usage of DSLs brings several benefits, such as:

- *encapsulation* – DSLs allow developers to encapsulate domain-specific knowledge in the language, making it easier to understand and use.
- *productivity* – DSLs can improve productivity by providing higher-level abstractions that are closer to the problem domain.
- *communication* – DSLs can improve communication between domain experts and developers by providing a common language for discussing requirements and specifications.
- *quality* – DSLs provide a way to express domain-specific constraints and requirements, which can help improve the quality of the software.

2.4.1 Internal and External DSLs

DSLs are classified as internal (or embedded [?]) and external. Internal DSLs are an idiomatic way of writing code in a general purpose programming language. They are not require a special parser or compiler, but they are limited by the syntax and semantics of the host language. Some programming languages, such as Lisp [?], Ruby [?] and Scala [?], intrinsically support the development of internal DSLs. External DSLs, on the other hand, are standalone languages that require a separate parser and compiler. They are more flexible and expressive than internal DSLs, but they are also more complex to develop and maintain.

In figure 2.3 we show an example of an internal DSL for arithmetic expressions in Scala. The DSL uses the Scala syntax to define arithmetic expressions, such as addition, subtraction, multiplication, and division. The DSL is embedded in the Scala programming language, allowing developers to write code that looks like a domain-specific language but is actually valid Scala code. It can be seen that no special parser or compiler is required to use the DSL, as it is written in Scala and can be executed using the Scala interpreter. Line 10 in 2.3 shows an example of using the DSL to define an arithmetic expression; the expression is syntact sugar for the Scala code `calc.add(5).multiply(3).subtract(1).divide(2)`.

In contrast, in figure 2.4 we show an example of an external DSL for arithmetic expressions in Scala. The DSL implementation is more complex than the internal DSL, as it requires a separate parser and compiler to process the DSL code. The DSL uses a custom syntax to define arithmetic expressions, such as `add`, `subtract`, `multiply`, and `divide`. The DSL is defined using a parser combinator library in Scala, which allows developers to define the syntax and semantics of the DSL in a declarative way.

2.5 Software and Language Product Lines

Variability in products is common in industrial production. For instance, in a car factory, a base car model can be customized with various options such as different colors, a navigation system, a cooling system, and parking sensors. This type of industrial production, known as variability-rich production, has been managed in traditional engineering environments through the development of product lines.

Software Product Line Engineering (SPLE) [?, ?] is a software engineering methodology that extends the concept of product lines to software development, following the dream of massive software reuse. The objective of SPLE is to find similarities among software products and to manage the variability in the software development process.

At the same time, a *software family* can be obtained using SPL [?] by defining a set of features. A feature can be a *core* feature, which is mandatory for all products in the family, or an *variable* feature, which can be included or excluded from the product. The set of features defines the *feature model* of the software family.

2.5.1 Feature Variability

Feature variability is a key concept in SPL. They are often developed using a *feature-oriented programming* [?] (FOP) paradigm, which allows developers to define features as

```scala
import scala.util.parsing.combinator._

class Calculator(val value: Int) {
  def add(x: Int): Calculator      = new Calculator(value + x)
  def subtract(x: Int): Calculator = new Calculator(value - x)
  def multiply(x: Int): Calculator = new Calculator(value * x)
  def divide(x: Int): Calculator   = new Calculator(value / x)
}

object CalcParser extends JavaTokenParsers {
  def number: Parser[Int] = wholeNumber ^^ { _.toInt }

  def factor: Parser[Calculator => Calculator] = (
    "add"      ~> number ^^ { x => _.add(x) }
  | "subtract" ~> number ^^ { x => _.subtract(x) }
  | "multiply" ~> number ^^ { x => _.multiply(x) }
  | "divide"   ~> number ^^ { x => _.divide(x) }
  )

  def expression: Parser[Calculator => Calculator] = rep(factor) ^^ {
    _.reduceLeft { (acc, f) => acc andThen f }
  }

  def parseExpression(input: String): Option[Calculator => Calculator] = (
    parseAll(expression, input) match {
      case Success(result, _) => Some(result)
      case _                  => None
    }
  )
}

@main def CalcExample =
  val calc = new Calculator(0)
  val expression = "add 5 multiply 2 subtract 1 divide 2"
  val resultFunction = CalcParser.parseExpression(expression)

  resultFunction match {
    case Some(f) => println(f(calc).
    case None    => println("Error parsing expression")
  }
```

Listing 2.4. *An external DSL for arithmetic expressions in Scala.*

first-class entities in the code. In this context, each feature represents either functional or non-functional characteristics of a subset of the software family's members. However, there is no universally accepted formal definition of software features. Typically, a feature is informally described as an increment [?]. Therefore, an SPL can be viewed as a collection of all available features, while a product configuration is a valid subset of the SPL. Each configuration corresponds to a specific member of the software family. Although various approaches to expressing SPLs exist in the literature, **feature models** [?] (FMs) are considered the *de facto* standard for variability modeling [?]. An FM is a tree-like structure that represents the features of a software family and their relationships. The root of the tree represents the software family, and the node represents a feature. The edges represent the relationships between the features.

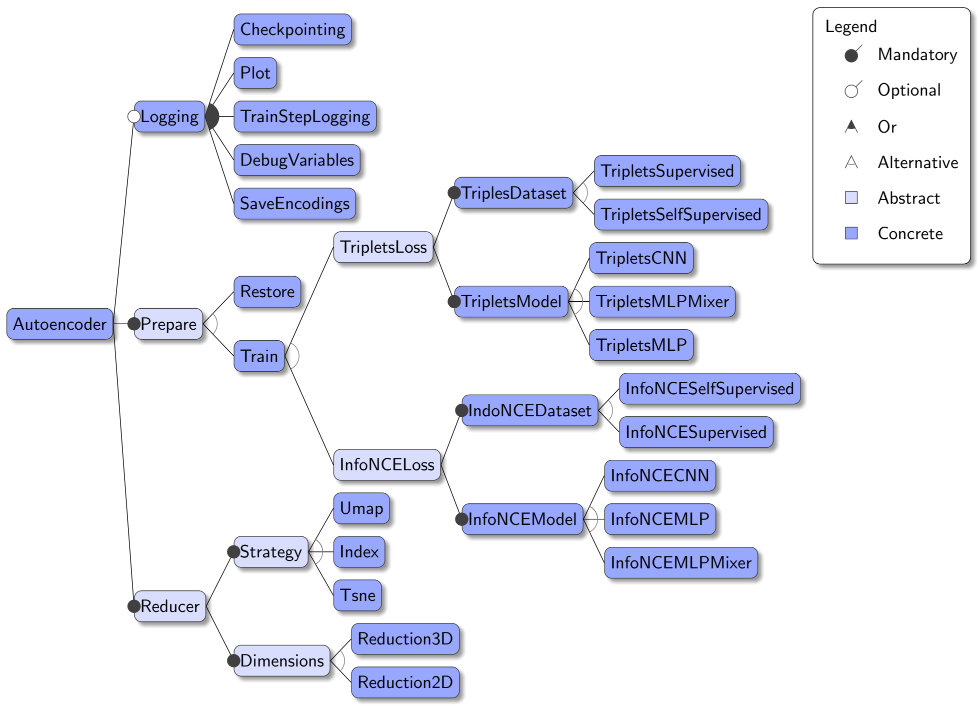


Figure 2.6. *FM of a family of neural networks used to encode the MNIST dataset, taken from [?].*

It is important to note that, given a FM, a software variant is identified through a configuration, which is a set of features that are selected from the FM. A configuration is valid if it satisfies all the constraints defined in the FM.

Figure 2.6 shows an example of an FM for a family of neural networks used to encode the MNIST dataset. FM structures implicitly define dependencies among features. Features can be *abstract* (`Reducer` and `Prepare` in Fig 2.6) or *concrete* (`Logging` and `Train` in Fig 2.6). An abstract feature represents a group of related features, while a concrete feature represents a specific functionality. Features are *mandatory* – such as `Dimension` in Fig 2.6 – if they must be included if their parent feature is selected, *optional* if they can be included or excluded. Features that are part of a group can be put in an *alternative* or *or* relationship.Or-relationship (children of Logging node in Fig. 2.6) means that only one of the features in the group can be selected, while an alternative relationship (children of Train node in Fig. 2.6) means that at least one of the features in the group must be selected if the parent feature is selected.

Dependencies can be explicitly defined using cross-tree constraints – i.e., Boolean expression among features of the FM [?], where each features represents a term in the expression. Each term is true if the corresponding feature is active in the current configuration and false otherwise. Cross-tree constraints support general Boolean operators such as AND, OR, NOT, IMPLIES, and IFF with their traditional meanings. The expressiveness of these constraints allows for defining dependencies that span the entire feature model, rather than being limited to parent-child relationships. A configuration is considered invalid if any cross-tree constraint evaluates to false.

An example of a cross-tree constraint is the following, where ⇒ represents the implication operator:

$$\texttt{Train} \Rightarrow \texttt{Logging} \vee \texttt{Strategy}$$

Implicit and explicit dependencies among features can lead to issues such as dead features (features that can never be activated), false-optional features (features labeled as optional but are actually required), and atomic sets (groups of features that must either all be activated or all be deactivated in any given configuration). Conducting static analysis of Feature Models (FMs) to identify these anomalies can enhance the quality of Software Product Lines. This research area encompasses both structural [**?**] and behavioral [**?**] methodologies. Additionally to FOP, other paradigms for SPLE such as aspect-oriented programming (AOP) [**?**] and delta-oriented programming (DOP) [**?**] exist.

2.5.2 Language Product Lines

Language Product Lines (LPLs) [**?**, **?**, **?**, **?**] are a specialization of SPLs that focus on the variability of programming languages. Researchers and practitioners have gained interest in LPLs [**?**, **?**, **?**] due to the increasing complexity of software systems and the need for more efficient and effective ways to manage variability in programming languages. LPLs are used to manage the variability in the syntax and semantics of programming languages. In particular, a family of programming languages can be defined using an LPL, where each language in the family is a member of the software family. The variability in the family is managed using features, which represent the different syntax and semantics of the languages in the family.

LPLs enable developers to efficiently generate and maintain multiple language variants within a coherent framework. In this context, any language workbenches, such as Neverlang, adopt the LPL approach to manage the variability [**?**, **?**] and the modularization of the syntax and semantics of programming languages.

Since LPLs are an engineering methodology for creating languages, researchers have also concentrated on assessing the quality of these LPLs. This assessment involves proposing various software metrics [**?**]. Additionally, there is an emphasis on mutation operators for mutation testing of LPLs, as outlined in [**?**].

Listing 2.5 shows an example of a Neverlang module for the addition expression. The module defines the syntax and semantics of the addition expression in the `ExprLang` language. The module contains a reference syntax, which defines the syntax of the addition expression, and a role, which defines the semantics of the addition expression. In the reference syntax bock, using the **provides** keyword, the module defines the features exported by the module. The keyword **requires**, instead, is used to import features from other modules, defining indirectly the dependencies among modules.

In Figure 2.7, we show an example of an FM for a family of expression languages. In the figure the *cross-tree constraints* are shown as dashed lines connecting the features. The cross-tree constraints are used to define dependencies among features that are

```
module AddExpr {
    reference syntax {
        provides { AddExpr: expression, numbers, sum, sub; }
        requires { MulExpr; }
        [ADD_0] AddExpr <- MulExpr;
        [ADD_1] AddExpr <- AddExpr "+" MulExpr;
        [ADD_2] AddExpr <- AddExpr "-" MulExpr;
    }
    role(evaluation) {
        [ADD_0] @{ $ADD_0[0].value = $ADD_0[1].value; }
        [ADD_1] .{ /*...*/ }.
        [ADD_2] .{ /*...*/ }.
    }
}
slice AddExprSlice {
    concrete syntax from AddExpr
    module AddExpr with role evaluation
}
language ExprLang {
    slices AddExprSlice UnaryExprSlice
           /* ... */
    roles syntax < evaluation
    rename { MulExpr -> UnaryExpr; }
}
```

Listing 2.5. *Example of a Neverlang module for the addition expression.*

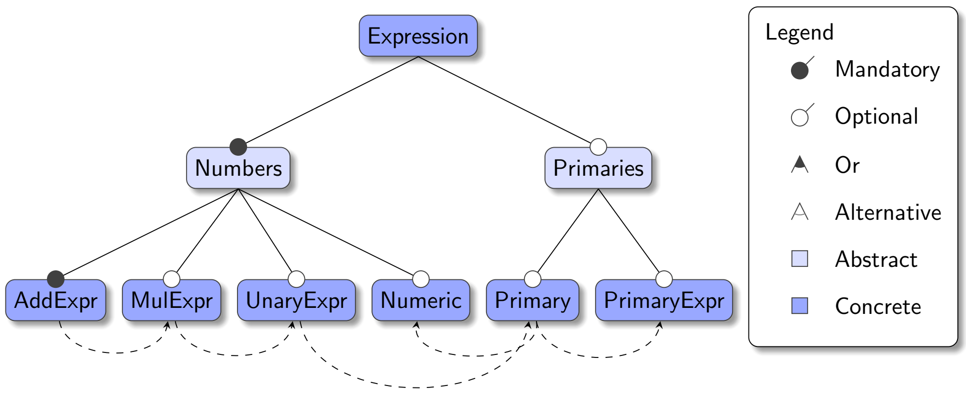


Figure 2.7. *Feature Model for the expression language, taken from [?].*

not explicitly defined in the FM. For instance, the cross-tree constraint between the `AddExpr` and `MulExpr` features states that if the `AddExpr` feature is selected, then the `MulExpr` feature must also be selected, for more details see [?]. Anyway, using the code in Listing 2.5 it it trivial to see the cross-tree constraints from `AddExpr` node to `MulExpr`, because of the `requires` keyword in the reference syntax block.

3 Related Work

According to Tomasetti [?], 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) [?]. To develop this kind of tool support, there are various language workbenches available that simplify the creation of abstract syntax, parsers, editors, and generators [?].

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 [?] 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 [?] 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) [?], 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 [?] which is based on Xtend[1].

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

3.1.3 MontiCore

MontiCore [?] 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 [?, ?] is a developement environment developed by JetBrains[2]. It is a *projectional editor* [?] 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 [?] 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 [?] (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 [?] 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) [?] for defining grammars. In Spoofax, the semantics is defined using the *Stratego* [?] language, which is a term rewriting language.Spoofax, the sew

3.1.7 Xtext

Xtext [?] 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 [?] parser and EMF [?] 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* [?], 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 [?]. Modularization and composition are critical and advanced features in language workbenches, enabling developers to

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

define a language by combining multiple features. Each feature can be developed independently and then integrated to create a new language, enhancing flexibility and reuse.

Developers typically manage 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 [**?**]. According to Bosch [**?**], a vertical decomposition still results in complex compilers or interpreters that are hard to maintain and extend, with components that are seldom reusable. He suggests that a vertical, functional decomposition should be accompanied by a horizontal, structural decomposition to better manage complexity.

A component can be any language feature (see Sect.2.5) 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[**?**] or not. Several language workbenches support modularization and composition of languages, either with precompiled components or without. Different language workbenches provide varying levels of support for these features. 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. Conversely, **Melange** supports composition through Encore models, which are declarative specifications of a set of classes that can contain fields, methods, and constraints. Encore models can be composed by renaming elements, allowing the developer to define a new language by combining multiple features. Another language workbench that supports modularization and composition of languages is **MontiCore**. It supports multiple grammar inheritance, but it does not support adding new attributes to imported symbols. **MPS**, being a projectional editor without a parsing phase, allows arbitrary notations to be combined. It supports language composition but does not support precompiled components. Moving to **Rascal**, rewrite rules are the principal mechanism for composing features, but 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* [**?**] to rewrite the tree, allowing the conversion from one AST to another. In contrast, **Xtext**, supporting only single grammar inheritance, does not support language composition.

The only language workbench that supports precompiled components is **Neverlang** [**?**]. While in **MontiCore** and **Xtext**, code implementing semantics can be excluded from the package, the grammar cannot be excluded, allowing for partial support for precompiled components.

Additionally, some language workbenches offer unique approaches to modularization and composition. For instance, **Spoofax** employs a meta-programming language, Stratego, which specializes in program transformation. This approach enables the composition of language features by transforming abstract syntax trees (ASTs), providing

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.*

a flexible mechanism for integrating various language aspects. On the other hand, **MPS** leverages its projectional editing paradigm to support modularization, allowing developers to mix and match different syntactic constructs seamlessly.

In Table 3.1, 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 indicate whether the language workbench supports, partially supports, or does not support modularization and composition, respectively. This overview highlights the varying capabilities of each tool, providing a clear comparison for developers looking to select a language workbench that best suits their needs.

3.3 IDE and LSP Generation

IDE generation is a crucial feature for language workbenches, as it enables developers to create tools that support the development of domain-specific languages (DSLs) within integrated development environments (IDEs). Nowadays, Language Server Protocol (LSP) is a key technology that enables the development of language servers that can be used in any IDE that supports the protocol. In this section, we will present some of the most relevant language workbenches that support IDE and LSP generation.

Four of the tools are closely integrated with IDE ecosystems. **Spoofax**, **Xtext**, and **Melange** are all Eclipse-based frameworks. Spoofax and Xtext support the automatic generation of plugins for text editor syntax highlighting and real-time visualization of the parsed AST (Eclipse's outline). Melange integrates with the EMF GUI-based model editor. In **MPS**, DSLs are both developed and used within the same IDE, with each concept (AST node type) having a default projectional editor component that can be optionally customized. **MontiCore** and **Rascal** can be used either as standalone command line programs or as Eclipse plugins. As detailed in [?], Section 4.5 and [?], respectively, both MontiCore and Rascal support the generation of Eclipse tooling for the languages developed using them.

On the other hand, LSP generation is a feature that is not supported by all language

Language Workbench	Native IDE gen.	LSP Gen.	LSP Mod.
JustAdd	○	○	○
Melange	3rd party (EMF)	○	○
MontiCore	●	○	○
MPS	●	○	○
Rascal	●	○	○
Spoofax	●	○	○
Xtext	●	●	○
Neverlang	●	●	●

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

workbenches. JustAdd, Melange, MontiCore, MPS, Rascal, and Spoofax all do not support LSP generation. **Xtext**, according to [?], is the only language workbench that supports the LSP.

Finally, also LSP modularization is a feature that is not supported by all language workbenches. JustAdd, Melange, MontiCore, MPS, Rascal. **Xtext**, as explained in section 3.2, does not support modularization and composition of languages, and therefore does not support LSP modularization.

In Table 3.2, we summarize the support for IDE and LSP generation of the language workbenches presented in this section. In the first column, we list the language workbenches. Using ●, ○, and ◐in the second, third, and fourth columns, we indicate whether the language workbench supports, partially supports, or does not support IDE generation, LSP generation, and LSP modularization, respectively.

4 Concept

4.1 The type system

In 2.3.1, we introduce the concept of type systems and we give a brief overview of *type checking* and *type inference*. Our goal is to have solid *application programming interfaces* (APIs) to build type systems for each language.

In this section, we will discuss the importance of the type system for our concept. What we are looking for in our type system is:

- **modularization**, that allows the definition of custom types and operations on these types, and the ability to combine them in a modular way;
- **flexibility**, since the type system is not known *a priori*, we need the ability to adapt the type system to the specific needs;
- **easy-to-use**, extending the default implementation of the type system with a new type, or implementing a new type from scratch should be easy and straightforward.

It is trivial to remember that the type system should also provide the basic functionalities of a type system, such as **type inference**, that allows the compiler to infer the type of an expression without the need to explicitly specify it; and **type checking**, that allows the compiler to check if the types of the expressions are correct.

Definition 1: type system item

given that the specifics of the target language for the reference type system are not predetermined (i.e., the language is not known *a priori*), we cannot assume the presence of standard programming constructs such as variables, functions, or objects. therefore, to maintain a broad and adaptable approach, we will use the term **item** to refer to any of these potential constructs or elements within the system. this ensures that our discussion remains relevant regardless of the particular features and paradigms of the target language.

4.1.1 type: the basic building block

The **Type** is the basic building block of the type system. It is used to represent the type of an expression, such as a variable, a function, or an object. The **Type** can be a primitive type, such as *int*, *float*, *string* or a custom type, such as a *class*, *interface*, or *enum*.

Definition 2: Type

The name **type** could be misleading because it does not exactly represent the type of an item but rather it is class of types that allows to build a specific type of a given category. Note that, this is not excluing that each concreate type could have a specific (class of) **type**, but defining a **type** as a class of types allows to define a generic **type** for all the types of a given category (i.e., all primitive types) and defining once the operations that can be performed on them.

In our concept, during the implementation of the type system, several types will be defined. Each type will have a set of operations that can be performed on it. In abstract terms, we can define a type as a set of operations that can be performed on it. Basically, each type should be answer to the following questions:

1. What is its **identifier**?
2. Can it be **assignable from** another type with a specific kind of **variance**?
3. Can it match a specific **signature**?
4. Does it need other types to be **bound**?

The second question is used to **type check** the *item* in the **Type System**. The type of **variance** is used to define the relationship between two types. The variance can be *covariant*, *contravariant*, or *invariant*.

Formal Definition 1 *Variance*
Suppose A and B are types, and $I[U]$ denotes application of a type constructor I with type argument U. Within the type system of a programming language, a typing rule for a type constructor I is:

- covariant if it preserves the ordering of types ($\leq$), which orders types from more specific to more generic: If $A \leq B$, then $I[A] \leq I[B]$;
- contravariant if it reverses this ordering: If $A \leq B$, then $I[B] \leq I[A]$;
- bivariant if both of these apply (i.e., if $A \leq B$, then $I[A] \equiv I[B]$);
- variant if covariant, contravariant or bivariant;
- invariant or nonvariant if not variant.

The third question is used to **type inference** the *item* in the **Type System**, simply answering to the question: can the type of the *item* match a specific **signature**? Finally, the last question is used to **bind** the type to its generic parameters. For example, the type *List<String>* is bound to the type *List<T>* with *needed types* being *String*.

4.1.2 Scope: the context of the type

Definition 3: Scope

The **Scope** is the context in which an identifier is defined, more precisely, where the binding between the type and an identifier is defined in the **Type System**. It is used to define the visibility and the lifetime of the identifier and its operations. The **Scope** can be, for example, a *local scope*, a *global scope*, or a *module scope*.

In our concept, a **Scope** can be see as a **Type** 4.1.1 that defines the visibility and the lifetime of the type or identifier and its operations. At this point, the **Scope** is not universal. In fact, for instance, *Java* do not categorize the **Scope**, such as *Block Scope*, as a **Type**. However, in order to achieve a representation of a *Block Scope* as a **Type**, we can define a **Scope** where at the questions

- Can it be assignable from another type with a specific kind of variance?
- Can it match a specific signature?

will answer *false*.

So, a **Scope** should be able to answer to the following questions, in addition to the questions that a **Type** should be able to answer:

1. What types are **visible** in the **Scope**?
2. Can a type be bound to an identifier in the **Scope**?
3. If exists, what is the **parent Scope**?
4. Can a **Scope** be marked as **external visible** and what items exported?

The first question is used to define the visibility of the type in the **Scope** and knowing what types are visible in the **Scope**. 4.1.5 will be used to answer this question. The second question is used to bind a type to the Scope. Every scope should have a set of rules that allow to determine if a type can be bound to the Scope. 4.1.4 will be used to answer this question. The third question is used to define the hierarchy of the Scopes. Trivially, this is represented as a tree. Finally, the last question is used to define the visibility of the Scope and its items. A **Scope** marked as *external visible* should be able to export some its items. This is useful during *type inference* phase, because whan searching for a type if a Scope (type) is marked as *external visible* the search should be performed recursively in all **items** exported by the **Scope**.

Now, we propose some examples of scopes in the world of object-oriented programming (OOP) languages. In OOP there are several types of scopes that determine the visibility and lifetime of items. The most common scopes are:

- **Class scope**: A class is a collection of variables and methods. Variables declared inside a class have class scope, and they can be accessed within that class and its instances. Methods declared inside a class also have class scope and can be accessed within that class and its instances.
- **Instance scope**: An instance is an object created from a class. Instance variables are declared inside a class, but outside any method, and have instance scope.

They can be accessed within that class's methods and any instance of that class.
- **Method scope**: Variables declared inside a method have method scope, and they can only be accessed within that method.
- **Block scope**: A block is a section of code enclosed in curly braces. Variables declared inside a block have block scope, and they can only be accessed within that block. This includes loops, conditional statements, and code enclosed in curly braces.
- **Static scope**: Static variables and methods belong to a class, not to an instance of that class. Static variables and methods have class scope and can be accessed using the class name without creating an instance of the class.

4.1.3 Signature: the definition of the type

A **Signature** is the definition of the type. It is used to define the structure of the type and to support *static typing* and *type checking*. In our type system, **each type** should have a Signature that defines the structure of the type.

Definition 4: Signature

The **Signature** in **structural type system** [?, ?] or in **nominal type system** [?] is the definition of the type. In a **structural type system**, the **Signature** is the structure of the type, while in a **nominal type system**, the **Signature** is the name of the type. The **Signature** is used also to support *static typing* and *type checking*.

A **structural type system** is a major class of type systems in which type compatibility and equivalence are determined by the type's actual structure or definition and not by other characteristics such as its name or place of declaration. A type system is **nominal** if compatibility and equivalence of data types is determined by explicit declarations and/or the name of the types.

A **Signature** should be able to answer to one question:

1. Can a type match the **Signature**?
2. Can I resolve the type of an identifier from the **Signature**?

Answering to this question can be simple or complex, depending on the type we would like to match with the **Signature**. For example, a type can be a primitive type, such as *int*, *float*, *string*, or a custom type, such as a *method* inside a *class*. For instance, a **Signature** is often used to define the type of a function. In this case, the **Signature** is the definition of the function in terms of its parameters and return type. Usually, every `Type` should have a `Signature` that defines the structure of the type. Additionally, to perform *type inference*, the **Signature** is used to infer the type of an expression.

4.1.4 Symbol Table Entry: an entry in the Typing Environment

A **Symbol Table Entry** is an entry in the **Typing Environment** (Symbol Table). It is used to store a informations associated to a symbol in the **Type System**.

Definition 5: Symbol

A **Symbol** is a name that represents a value, a variable, a function, or any other item in a program. In the **Type System**, a **Symbol** is used to represent a type, a variable, a function, or any other item in the program. Usually, symbols are collected during the *parsing* phase of the compilation process and stored in a **Symbol Table**. A symbol can be a *nonterminal symbol* or a *terminal symbol*. A *nonterminal symbol* is a symbol that can be expanded into other symbols, while a *terminal symbol* is a symbol that cannot be expanded further.

The **Symbol Table Entry** should be able to answer, at least, to the following questions:

1. What is the **Type** of of the associated symbol?
2. What is the **Kind** of of the associated symbol?
3. What is the **Location** of of the associated symbol?
4. What are the **Details** of of the associated symbol?

As one can guess, the **Symbol Table Entry** is used to store the information associated with a symbol in the **Type System**. Note that to answer the first question, the **Symbol Table Entry** should have a **Type** associated with it. The answer to the second question of 4.1.3 can help to infer the **Type** of the associated symbol if the **Type** is not explicitly defined. The second question is used to define the **Kind** of the associated symbol. The **Kind** can be, for example, a *declaration*, a *definition*, a *use* or an *import*. The third question is used to define the **Location** of the associated symbol. The **Location** can be, for example, a *line number* or a *column number*. Finally, the last question is used to define the **Details** of the associated symbol. The **Details** can be, for example, a *description* or *extra information* about the symbol.

Entry Type Binder: an helper to perform type binding

The **Entry Type Binder** is an helper to perform type binding in the **Type System**. It is used to bind a type to an identifier in the **Scope**. The **Entry Type Binder** should be able to answer to the following questions:

1. Is this identifier already bound to the **Scope**?
2. Can I bind an identifier to a `SymbolTableEntry` in this **Scope**?

The first question is used to determine if the identifier is already bound to the **Scope**. If the identifier is already bound to the **Scope**, the **Entry Type Binder** should return an error. The second question is used to determine if the identifier can be bound to a `SymbolTableEntry` in the **Scope**. If the identifier can be bound to a `SymbolTableEntry` in the **Scope**, the **Entry Type Binder** should bind the identifier to the `SymbolTableEntry`.

4.1.5 Typing Environment: the symbol table

The **Typing Environment** is the symbol table in the **Type System**. It is used to store the information associated with the symbols in the **Type System**. Basically, the **Typing Environment** is a map between symbol or identifier and `SymbolTableEntry`.

The **Typing Environment** should be able to answer to the following questions:

1. What are the symbols in the **Typing Environment**?
2. Can I get the `SymbolTableEntry` associated with a symbol in the **Typing Environment**?
3. Can I add a symbol to the **Typing Environment**?
4. Can I remove a symbol from the **Typing Environment**?

In section 4.1.2, we defined the **Scope**. Combining the **Scope** with the **Typing Environment**, we can define the visibility of the types in the **Scope** and the lifetime of the types in the **Scope**. Answering to the first question, the **Typing Environment** should be able to return the symbols in the **Typing Environment**; it it like answering to the question: what are the types or identifiers visible in the **Scope**? The second question is used to get the `SymbolTableEntry` associated with a symbol in the **Typing Environment**. This is useful to get the information associated with the symbol. The third question is used to add a symbol to the **Typing Environment**. This is useful to add a new symbol to the **Scope**. Finally, the last question is used to remove a symbol from the **Typing Environment**. This is useful to remove a symbol from the **Scope**.

In Figure 4.1, we have a final representation of the type system.

4.1.6 Type Inference: the ability to infer the type of an expression

As already mentioned in 2.3.4, **Type Inference** is the ability to infer the type of an expression without the need to explicitly specify it. It is used to determine the type of an expression based on the context in which it appears. In our concept, **Type Inference** is used to infer the type of an expression in the **Type System**. The **Type Inference** should be able to answer to the following questions:

1. Can I infer the type of an expression?
2. Can I resolve the type of an identifier from the **Typing Environment**?

The first question is used to determine if the type of an expression can be inferred. If the type of an expression can be inferred, the **Type Inference** should infer the type of the expression. The second question is used to resolve the type of an identifier from the **Typing Environment**. This is useful to infer the type of an expression based on the context in which it appears.

There are multiple algorithms to perform **Type Inference**, such as *Hindley-Milner* [**?**, **?**], *unification-based* [**?**], and *constraint-based* [**?**] algorithms. In our concept, we will use a `Strategy Pattern` to define the **Type Inference** algorithm. In order to perform **Type Inference**, we should have a list of all possibile candidates for the type of the expression. This list creation is performed by the **Type Lookup** algorithm 1.

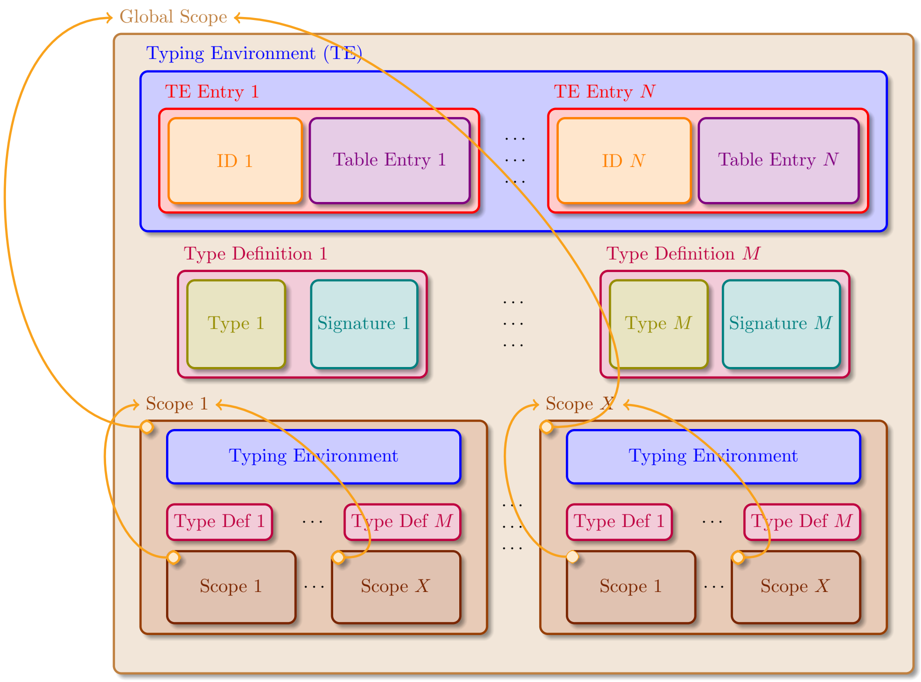


Figure 4.1. *A final representation of the type system.*

Algorithm 1 Pseudocode to show how type lookup is performed

```
Require: $I$ identifier, $SIG$ signature, $S$ scope
Ensure: $L$ list of all candidates
  $L \leftarrow \emptyset$                                   ▷ Initialize $L$ with an empty list
  $CS \leftarrow S$
  while $CS \neq \emptyset$ do
      for each $Type$ in $CS$ with identifier $I$ do
          if $Type$ match signature $SIG$ then
              $L \leftarrow L \cup \{Type\}$
          end if
      end for
      $CS \leftarrow$ parent of $CS$                             ▷ parent could be $\emptyset$
  end while
  return $L$
```

4.1.7 Example: a simple C program

Now, we propose an example where putting all the concepts together. In Listing 4.1, we have a simple C program that defines a global variable `const_one` and a function `main` that prints (using the `printf` function) the value of the variable `const_one` and the value of the variable `two`.

```c
#include <stdio.h>
const int const_one = 1;
int main () {
    int two = 2;
    printf("Three: %d", const_one + two);
    return 0;
}
```

Listing 4.1. *Example of a simple C program.*

Scope	Identifier	Type	Kind	Location	Details
Global	stdio.h	header	import	1:1 - 1:18	
Global	const_one	int	declaration	3:11 - 3:19	const
Global	main	function	declaration	5:5 - 5:8	
main	two	int	declaration	6:9 - 6:11	
main	printf	function	use	7:5 - 7:10	external
main	const_one	int	use	7:25 - 7:33	
main	+	function	use	7:35 - 7:35	
main	two	int	use	7:37 - 7:39	

Table 4.1. *Language Workbenches Supporting Modularization, Composition and Precompiled Features.*

After parsing the program, the **Typing Environment** should contain the following information:

- The `stdio.h` header is imported in the **Global Scope**.
- The `const_one` variable is declared in the **Global Scope** as a constant integer.
- The `main` function is declared in the **Global Scope**.
- The `two` variable is declared in the `main Scope` as an integer.
- The `printf` function is used in the `main Scope` and is marked as external.
- The `const_one` variable is used in the `main Scope`.
- The + operator is used in the `main Scope`.
- The `two` variable is used in the `main Scope`.

Note that in table 4.1, we have the **Scope** as first column, but in reality, is the **Scope** that defines the visibility and the lifetime of the types in the **Typing Environment**. In fact, in our explanation in section 4.1.2, we defined the **Scope** as the context in which an identifier is defined, more precisely, where the binding between the type and an identifier is defined in the **Type System**. So, ideally, every **Scope** should have a **Typing Environment** associated with it. As we can see, these information will be used also to perform the **modularization** of the LSP (see section 4.2).

4.2 Towards a modular type system

In this section, we will discuss the importance of having a modular type system. In particular, we will focus on the ability to define types and operations on these types in a modular way and the ability to combine them. In the context of language workbenches, the ability to define types and operations on these types in a modular way is crucial. In fact, the language workbenches should be able to support different languages and paradigms, and the ability to define types and operations on these types in a modular way is crucial to achieve this goal.

The **modularization** of the type system is important for several reasons:

- **Reusability**: The ability to define types and operations on these types in a modular way allows to reuse the types and operations in different contexts.
- **Extensibility**: The ability to define types and operations on these types in a modular way allows to extend the types and operations with new types and operations.
- **Flexibility**: The ability to define types and operations on these types in a modular way allows to adapt the types and operations to the specific needs.

In the context of language workbenches and Language Product Lines (SPLs), we are trying to aswer to the following research questions:

RQ1 How can we define types and operations on these types in a modular way?

RQ2 How can we combine types and operations on these types in a modular way?

RQ3 Can we achieve the type checking and type inference in a modular way?

Thus, everything we have seen so far is very abstract, in the following sections we will see how the following APIs are adapted to an AST.

Consider the simple JavaScript program in Listing 4.2, which defines two functions to sum two numbers.

```
function sum1(x) {
    return sum(x, 1);
}

function sum(x, y) {
    return x + y;
}
```

Listing 4.2. *A simple JavaScript program that defines two functions to sum two numbers..*

The AST of the program in Listing 4.2 is shown in Figure 4.2.

4.2.1 Compilation Unit: a logical unit of source code

Usually, a **compilation unit** is a logical unit of source code that is processed by the compiler as a single entity. It typically consists of a single source code file and any header files that it includes. In some programming languages, such as C and C++,

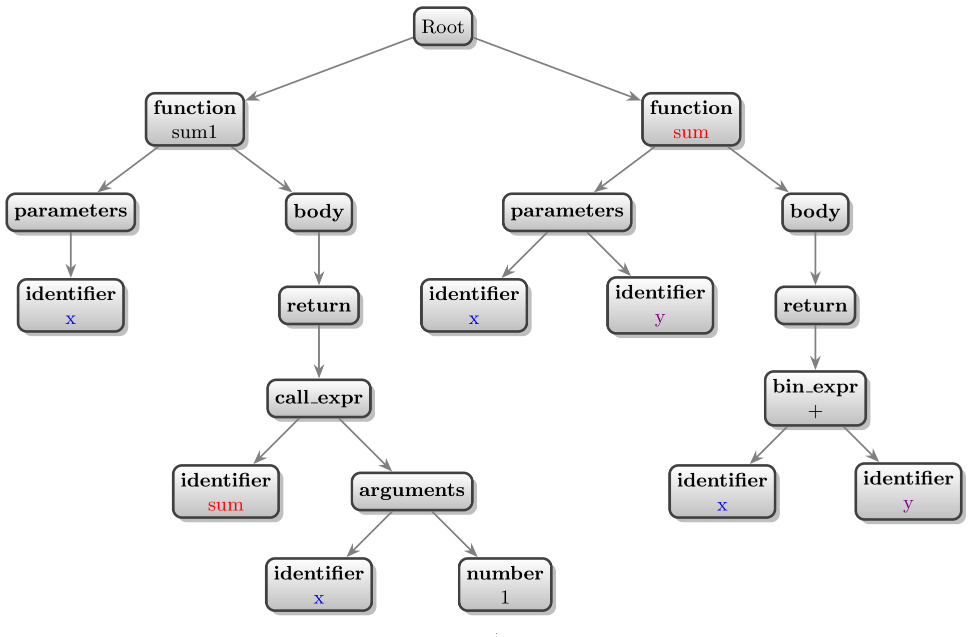


Figure 4.2. *AST of source code 4.2..*

each compilation unit (also kwown as translation unit) must have a unique name and can contain multiple functions or variables. Other programming languages, such as Java [?], use a different approach where each compilation unit corresponds to a single class definition.

Definition 6: compilation unit

In this particular concept, a **compilation unit** is responsibile for a specific portion of the code, which includes a set of subtrees within the abstract syntax tree (AST). It is responsible for traversing the nodes of the AST that pertain to its assigned portion through a compilation unit task (see more 4.2.2). Additionally, it serves as a supportive tool for semantic actions, where the compilation unit is utilized to execute detailed analysis for performing certain actions, as explained in further detail below.

The operations for which a compilation unit is used are:

- **enter or leave a scope**, when a new compilation unit is created, it has a stack with a scope that represents the initial scope.
  - when *entering* a scope it is pushed onto the stack and the old scope becomes the parent of the new scope (e.g. when transitioning from a global scope to a function scope, the global scope becomes the parent of the function scope. This enables all the entities that exist within the global scope to still be inferred from the function scope).

    - when *exiting* the current scope is popped from the stack.
- **bind an identifier to an entity** on the current scope
- perform **type inferences** on the current scope

There are several actions that are performed during the traversal of the AST. Supposing we have the AST 4.2, the sequence of events during its preorder traversal would be as follows:

1. creating a new compilation unit with `global` scope as initial
2. binding a new function with identifier *sum1* in the current scope
3. entering in scope of the function *sum1*
4. evaluating ast node *params* and *body*
5. exit from scope of the function *sum1* and returning inside global scope
6. binding a new function with identifier *sum* in the current scope
7. entering in scope of the function *sum*
8. evaluating ast node *params* and *body*
9. exit from scope of the function *sum* and returning inside global scope

When the body of the sum1 function (`return sum(x,1)`) is evaluated at point 4, it will performed the following operations:

- infer the type of x and 1
- and then infer the type of sum given type of x and type of 1 as signature

In the first case the inference is **successful** because *x* has been defined before (in the ast node params) and 1 is a primitive type known a priori in this case it will be up to the developer to find a way to store the entities that make up the base of the language.

On the other hand, the inference of the *sum* type **fails** because the relevant node, which defines the function, has not yet been visited. To address this issue, two potential solutions are available:

- one option is to mandate that users define functions before using them, potentially through the introduction of concepts similar to C function prototypes,
- another solution involves altering the order of visits, ensuring that all functions are defined before entering their respective bodies.

The choice that was made to maintain the flexibility of the API was to modify the visit order of the AST, the next section explains how the visit order is managed.

4.2.2 Compilation Unit Task: a task to perform compilation

As discussed in the earlier section, the preorder visit approach may encounter challenges when trying to infer types that have not been defined yet. To address this issue, the programmer can determine the order of node visitation. This allows the option to either visit a node immediately or create a compilation unit task that will execute the visit only when authorized (further details to follow).

The programmer has the flexibility to determine the sequence of node visits in the following ways:

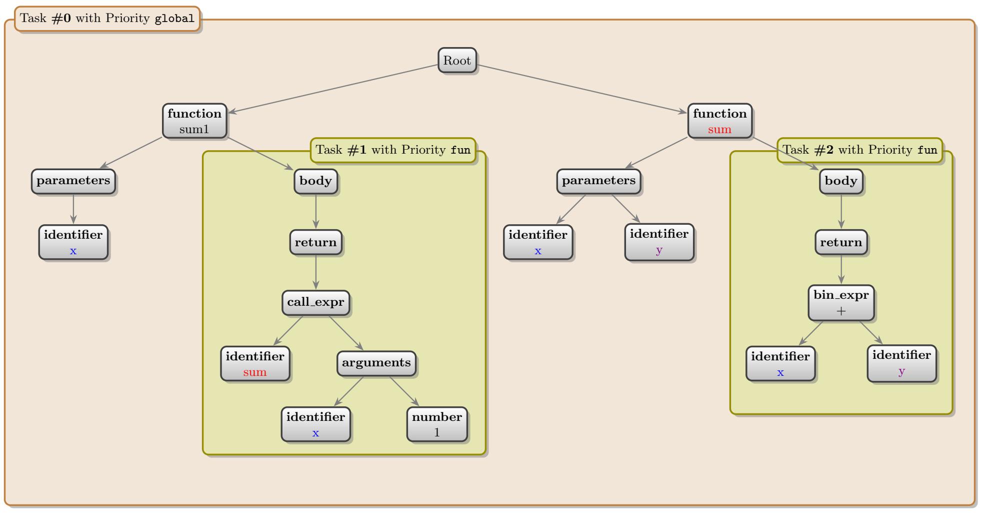


Figure 4.3. *AST with **task** and **priority** annotations..*

– Immediate node visit: The programmer can choose to visit other nodes without delay.
– Compilation unit task creation: The programmer can create a compilation unit task that visits associated nodes only when permitted.

These operations can coexist, allowing the programmer to visit certain nodes immediately while deferring others as compilation unit tasks. Each task has its own dependency and priority, which are crucial factors in determining the execution order. For instance, while at node "function sum1" in tree 4.2, the programmer can opt to visit the "params" node right away, while creating a compilation unit task for the "body" node.

The programmer specifies points in the code where a compilation unit task is created, enabling deferred traversal of specific nodes in the Abstract Syntax Tree (AST). To ensure smooth traversal, the compilation unit task must adhere to the following properties:

– It must be associated with exactly one Compilation Unit.
– Multiple unit tasks cannot visit the same node simultaneously.
– Each task can only be executed once during compilation.

Remarking, each compilation unit task must be associated with a single compilation unit, meaning that whenever a task is created, a corresponding compilation unit must also be created.

After introducing the concept of a compilation unit task, the visit order of the AST nodes can be reviewed as follows (see figure 4.3):

1. When the root node is visited, a task (#0) associated with the root of the AST node is created.
2. Task #0, when executed, performs a preorder visit of the children nodes (function sum1 and function sum). The only difference compared to before (the visit done

in 4.2) is that the body node is not executed immediately. Instead, another task is created for the body node, which will be executed subsequently.

3. Tasks #1 and #2 are executed only after task #0 has finished, ensuring that the inference of "sum" can be correctly performed because it happens after the definition of the function "sum".

However, the method for managing the order of task execution has not been clarified yet. In the next section, we will discuss how this will be accomplished.

4.2.3 Compilation Helper: an helper to perform compilation

A compilation unit is associated with a piece of code, whereas a compilation helper is shared among all the compilation phases. The compilation helper is responsible for creating and registering new compilation units, along with their associated tasks. Later, the registered tasks can be executed by the compilation helper.

Root type and root compilation unit

To initialize the compilation helper, it is required to define a root type that represents the global scope of the entire source set being compiled. Using this global scope, the compilation unit root is created, and its associated task will be the first one to be executed.

Hooks

Compilation Helper provides the capability to define custom hooks that are executed at specific phases during the execution process. The available hooks currently include:

- `beforeAll`: which is executed prior to the initialization of all roots
- `beforeEach`: which is executed before the initialization of each individual root in every file

These hooks can be easily implemented by overriding the default methods provided by the Compilation Helper.

Task creation

In order to generate a task, it is imperative to establish a compilation unit. When you are creating a compilation unit the following situations can arise:

- *Creating a new scope*: This necessitates the generation of a new compilation unit that is linked to the newly created scope.
- *Utilizing the current scope*:
  - If the current scope is the root scope, the root task is updated with additional nodes to visit.
  - If the current scope is not the root, a fresh compilation unit is generated.

When a new compilation unit is generated, a corresponding task is also created with two key pieces of information: its dependency and execution priority. In our case, each task is dependent on the task from which it was generated, ensuring that it is executed only after its parent task has completed. The execution priority is a class `T` that extends `Comparable<T>`, allowing for custom prioritization of tasks.

Both the dependency and priority information are utilized by the executor to effectively manage the order in which the tasks are executed, as we will see in the next section.

Algorithm 2 The algorithm used for compilation task execution

```
Require: $X$ list of all prioritized tasks
  while $X \neq \emptyset$ do
      $N \leftarrow$ collect tasks with highest priority in $X$
      $X \leftarrow X \setminus N$
      create a DAG from $N$
      execute the DAG              ▷ DAG execution could populate $X$ with new tasks
      wait for completion
  end while
```

Task execution

The tasks are executed following the algorithm shown in 2. Tasks that are independent of each other can run in parallel.

4.2.4 A family of type systems

In the context of language workbenches and Software Product Lines (SPLs), the modularization of type systems enables the creation of a family of type systems. This is particularly advantageous for supporting various languages and paradigms. By constructing the type system in a modular fashion, each component or "artifact" of the type system can be activated or deactivated as needed, resulting in a tailored type system for specific language requirements. This section explores how such modularity facilitates flexibility, reusability, and extensibility in language development.

Modular Type Systems in Language Workbenches

Language workbenches aim to provide tools and frameworks to define and manipulate programming languages. One of the core aspects of a language is its type system, which encompasses the rules and operations for type checking and type inference. A modular type system allows the language designer to define these rules and operations in separate, interchangeable modules.
Key Benefits:

- **Reusability**: Modular type systems enable the reuse of type definitions and operations across different languages and contexts. For example, a module that defines numeric types and arithmetic operations can be reused in multiple languages without modification.
- **Extensibility**: New types and operations can be added to the system without altering the existing modules. This is particularly useful in evolving languages or when creating domain-specific languages (DSLs) that require specialized types.
- **Flexibility**: Modules can be selectively activated or deactivated, allowing for the customization of the type system to meet specific needs. This is essential for supporting multiple paradigms or language variants within the same framework.

Language Product Lines (LPLs)

In SPLs, the concept of modular type systems aligns with the idea of product families. Each "product" in the line can be seen as a specific configuration of the language, with certain type system features enabled or disabled. Creating a Family of Type Systems

- **Type System Artifacts**: Each component of the type system (such as type definitions, type checking rules, and type inference mechanisms) is treated as an artifact. These artifacts can be independently developed, maintained, and combined.
- **Configuration Management**: The language workbench provides mechanisms to manage configurations, allowing language designers to specify which artifacts are included in a particular language instance. This can be achieved through configuration files, annotations, or a graphical interface.
- **Dependency Management**: Dependencies between artifacts are managed to ensure that the activation of one artifact automatically includes any required dependencies. For instance, enabling a type inference module may also require certain type definitions to be included.
- **Dynamic Activation**: During compilation or interpretation, the system dynamically activates the relevant type system artifacts based on the configuration. This allows the same underlying infrastructure to support different language features as needed.

Example: Type Checking and Inference in Action

Consider a scenario where a language product line includes multiple variants of a programming language, each with different type-checking requirements. One variant might require strict type checking for all variables, while another might allow more lenient type inference for certain constructs. Using a modular type system, these variants can be configured by selectively activating the appropriate type-checking and inference modules.

For instance, the type-checking module for strict type enforcement can be activated for the strict variant, ensuring that all variables must be explicitly typed. In contrast, the lenient variant can activate a module that implements type inference rules, allowing the system to deduce types where explicit annotations are missing. Additionally, common modules that handle basic type definitions and generic type operations can be shared across both variants, promoting reuse and reducing redundancy.

```java
public static void main(String[] args) {
    Language language = // Target language produced by the Language Workbench
    String sourceCode = // Source code written by the user
    try {
        language.parse(sourceCode);
    } catch (ParseException e) {
        // Send diagnostic information to the language client
    }
}
```

Listing 4.3. *Example of catching a Syntax Error in Java.*

4.3 The relevance of the type system in the LSP design

The type system is the core of the LSP design. We illustrate the need of having a type system by focusing on the ability to respond to requests from a *Language Client*.

In the reminder of this section, we will evaulate the relevance for three of the most important LSP feautre introduced in 2.1.3.

Diagnostic Analysis

Currently, the LSP provides the ability to perform *diagnostic analysis* on the code. This feature is useful to provide feedback to the user about the correctness of the code. In compilers design, a *Diagnostic* can be produced by different phases of compilation, such as the *lexical analysis*, *syntax analysis*, and *semantic analysis*. The **Syntax Errors** are detected by the *lexical analysis* and *syntax analysis*, usually during the *parsing* phase. The **Semantic Errors** are detected by the *semantic analysis*, usually during the *type checking* phase. In modern compilers, an additional phases can be added to the compilation process, *data flow analysis* and *control flow analysis*, that can be used to detect more complex errors, such as *unreachable code* or *dead code*.

In Language Workbenches world, usually it is common to have an instance of a **Language** and a **Source Code** that should be parsed and analyzed by the Language. Taking into account the **Syntax Errors**, assuming that the **Language** is able to parse the **Source Code** and the **Language** is able to provide errors and warnings during the *parsing* phase, should be easy to provide the *Syntax Errors* to the *Language Client* (see Listing 4.3).

The **Semantic Errors** and **Data Flow Analysis** are more complex to implement. In fact, in order to detect a *Semantic Error*, the **Language** should be able to perform *type checking* on the **Source Code** in order to verify that the code can be executed without any unexpected errors. The *Data Flow Analysis* is necessary to understand if the code is reachable, and to do this, the **Language** should be able to perform static analysis on the **Source Code**.

Jump to Definition

The ability to access the definition of a symbol, such as a function or a variable, is a common feature in modern IDEs. To enable this functionality, a **Symbol Table** is required and should be able to map a given row and column position in the source code to the corresponding symbol and its definition. During this phase, the typechecking is required to bind the symbol to its type. This is necessary to provide the correct definition of the symbol to the user. In addition to the symbol table, the **Language** should be able to provide the *Scope* of the symbol, in order to understand if the symbol is visible in the current context.

Code Completion

To effectively handle this kind of request, such as the one depicted in figure 2.5, the language server needs to comprehend the type of symbol for which suggestions are to be provided. Once the type is determined, returning relevant suggestions becomes straightforward. However, a challenge arises when suggestions must be presented to the user while they are still writing, as the source code may be syntactically incorrect during the writing process. To address this issue, a parser with error detection and recovery [?] capabilities is required to enable effective suggestion generation despite the presence of syntax errors.

4.4 LSP: a modular approach

LSP and DAP are protocols that describes a common *Application Programming Interface* (API) that the **language server** (LS) should implement, with the benefit of having only one implementation of the LS and multiple clients (IDEs and SCEs) that can consume it, essentially establishing a *client-server* relationship through a communication channel (e.g., *pipes* or *sockets*). However, the implementation of an LS and its integration with an IDE/SCEs is still a complex task, as it requires the knowledge of the LSP specification and the implementation of the language support. The implementation [?] of an LS is done entirely manually and it is a *top-down* activity, where most of the time is spent on the design and implementation data structures and algorithms. Since 1990s [?], researchers have started talking about the *Software Product Lines* (SPLs) [?, ?] to move towards a more modular world, where the implementation of a software system can be done in a compositional way, by composing the features of the SPL. When a SPL is applied to the implementation of a programming language, each product corresponds to a language variant [?] taking the name of *Language Product Lines* (LPLs) [?]. LPLs have been successfully used in both GPLs [?, ?, ?] and DSLs [?, ?, ?].

What I want to prove with this thesis is that the implementation of an LS could be a *bottom-up* activity, where each LSP or DAP functionality can be seen as a separate *feature module* [?, ?] splitted across the language artifacts, where each artifacts can be part of one or more *language features* (see Figure 4.4). These units can be composed to provide a modular implementation of the LS. This approach is supported by the fact that the

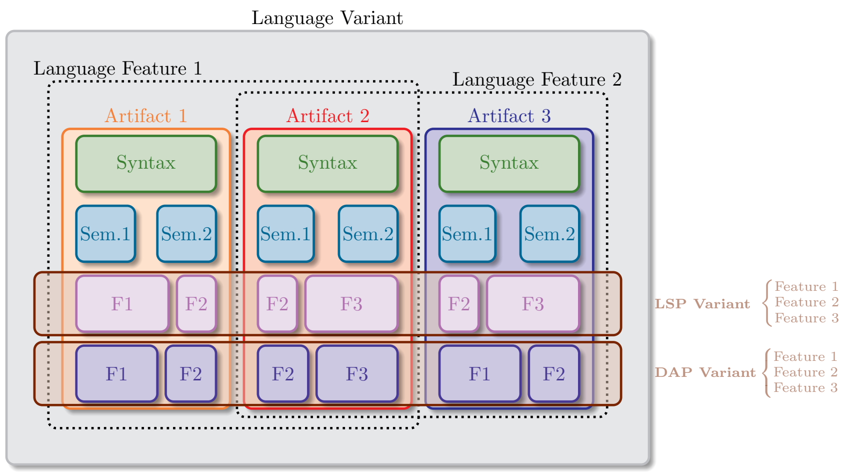


Figure 4.4. *Proposed approach to modular implementation of LSP and DAP..*

LSP and the DAP are *language-agnostic* protocols [?, ?] (see Fig. 2.1), which means that it does not impose any restrictions on the implementation of the LS, as long as it respects the specification of the protocol. In *feature-oriented programming* (FOP) [?, ?, ?], a feature module is a unit of composition that encapsulates a specific functionality, and it is a first-class entity that can be composed with other feature modules to form a software system; similar to an aspect module that encapsulates a crosscutting concern in *aspect-oriented programming* [?, ?, ?]. Using FOP in language development, a family of languages [?] can be defined by composing feature modules [?], and a language can be seen as a product of the family. In this way, the implementation of the LS is a *bottom-up* activity, where each artifact has attached a part of LSP and DAP feature module that implements the LS functionality for that *language fragment*, and these units can be composed to provide a modular implementation having **variants** of the LS.

4.4.1 Index Tree

The Index Tree is a sophisticated data structure specifically designed to store and manage information about the source code, such as symbols, types, and references. This structure is pivotal for Language Servers (LS) to efficiently provide essential features like "Jump to Definition" and "Code Completion." Its design facilitates rapid and accurate symbol location based on their positions within a text file, making it an integral component of modern code editors and development environments. In figure 4.5, we illustrate the structure of the Index Tree and its key components; as example, we consider the source code in Listing 4.2.

The Index Tree is structured as follows:

- **Nodes and SymbolTableEntry:** Each node in the Index Tree is represented by a `SymbolTableEntry`. This entry encapsulates all relevant information about a particular symbol in the source code, including its type, position, and references.

Algorithm 3 The algorithm used to get a symbol from IndexTree

```
Require: $(row, col)$ a position on the file, $N$ the node of the index tree
Ensure: A set of SymbolTableEntry entries
  $S \leftarrow$ the SymbolTableEntry entry associated with $N$
  if $(row, col)$ is within the selection range of $N$ then
      return $\{S\}$
  else if $(row, col)$ is within the folding range of $N$ then
      for each C in $N$'s children do
          $Result \leftarrow$call this algorithm with $(row, col)$ and $C$
          if $Result$ is not empty then return $Result$
          end if
      end for
  else
      return $\emptyset$
  end if
```

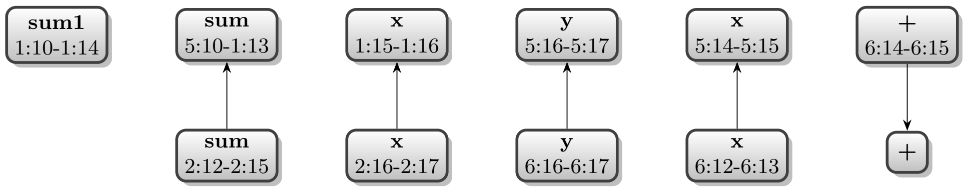


Figure 4.5. *Index Tree structure for the source code in Listing 4.2..*

- **Internal Nodes and Folding Range:** Internal nodes in the tree must possess a folding range. This range is a continuous segment within the source code that encompasses all the symbols contained within the node. The folding range ensures that related symbols are grouped together logically.
- **Selection Range:** The selection range refers to the exact range within a node that contains a specific symbol. It is more precise than the folding range and is used for pinpointing the location of a symbol within the node.

The Index Tree supports several critical functionalities for the Language Server:

- **Efficient Symbol Search:** The tree structure allows for straightforward symbol searches. Given a position in the text file, the tree can be traversed to locate the corresponding node, thanks to the hierarchical organization of folding ranges and selection ranges.
- **Jump to Definition:** By leveraging the Index Tree, the Language Server can quickly locate the definition of a symbol when a user invokes the "Jump to Definition" feature.
- **Code Completion:** The tree also aids in code completion by providing a structured and easily navigable repository of all symbols and types within the source code.

One of the key advantages of the Index Tree is its modularity:

– **Language-Specific Populations:** The tree is populated by the specific programming language, meaning each language can add its own symbols, types, and references to the tree.
– **Composition with Other Features:** The Index Tree serves as a foundational component that can be composed with other language features. This modular approach allows for a more flexible and scalable implementation of the Language Server, enabling it to support a wide range of languages and features.

The efficiency of the Index Tree lies in its hierarchical structure and the way it organizes symbols within ranges. The time complexity for inserting a symbol into the tree or searching for a symbol generally depends on the depth of the tree. In the average case, both operations can be expected to run in $O(\log n)$ time, where n is the number of symbols in the source code. This logarithmic complexity ensures that the Index Tree can handle large codebases efficiently, providing quick access to symbols even as the size of the source code grows.

In summary, the Index Tree is a highly efficient and modular data structure that plays a crucial role in modern Language Servers. Its ability to store and organize symbols, types, and references, combined with its efficient search and retrieval capabilities, makes it an indispensable tool for providing advanced code navigation and completion features.

4.4.2 Symbol Dependency Graph

The symbol dependency graph serves as a vital data structure within the language server framework, designed to interconnect symbols in a manner that facilitates efficient responses to client queries. In essence, it's a graph where nodes represent SymbolTableEntry instances, and edges are annotated with labels denoting the type of relationship. Currently, the primary relationship label is "usage," indicating that one symbol uses another.

This graph plays a crucial role in several key functionalities of the language server:

– **Calculating Symbol References**: By traversing the graph, the server can determine all references to a specific symbol throughout the codebase. This is essential for tasks like finding where a symbol is used or referenced.
– **Navigating to Symbol Definitions**: Utilizing the graph, the server can pinpoint the exact location within the code where a symbol is defined. This capability is crucial for enabling developers to quickly navigate to the declaration of a symbol.

Diagnostic Handling

During the parsing and static analysis phases, the language server gathers diagnostics, which are issues or potential problems identified in the code. These diagnostics are collected and presented to the client at the end of compilation. Custom diagnostics can be registered through exceptions or specialized data structures like **TypesystemException** or `LogRecord`.

Definition, References, and Hover Features

When a user requests information about a specific position in a file—such as the definition, references, or details on hover—the server follows these steps:

- **Index Tree Lookup**: Using an index tree, the server identifies the symbol associated with the requested position, if it exists.
- **Graph Traversal**: For definition and reference queries, the server traverses the symbol dependency graph to retrieve all relevant definitions or references of that symbol. This traversal is guided by the relationships defined in the graph.
- **Hover Information**: For hover queries, the server executes methods annotated with @Hover to provide detailed contextual information about the symbol at the specified position.

Document Symbol, Semantic Token, Inlay Hint, and Folding Range

These capabilities involve listing and presenting symbols and structural elements within a file:

- **Document Symbol**: Lists all symbols present in a file using the index tree.
- **Semantic Token**: Provides structured information about tokens in the code for language-specific operations.
- **Inlay Hint**: Displays subtle UI hints, annotations, or metadata within the editor.
- **Folding Range**: Enables collapsing and expanding regions of code based on predefined folding ranges associated with symbols.

Additional Capabilities

Other functionalities, though not implemented by default, can be achieved by leveraging data structures obtained during static analysis. These capabilities include advanced tasks such as semantic analysis, type checking, and code optimization, which benefit from the comprehensive understanding of code structure and dependencies provided by the symbol dependency graph.

In summary, the symbol dependency graph is pivotal in enabling precise navigation, comprehensive symbol analysis, and the delivery of rich contextual information within the language server ecosystem. Its structured representation of symbol relationships forms the backbone for many essential IDE features, enhancing developer productivity and code comprehension.

4.5 A *modular* DSL for semplifying the LSP and type system development

```
           ⟨program⟩  ⊨  ⟨type definition⟩*⟨scope definition⟩*⟨type inferencing⟩*
                         ⟨type checking⟩*⟨error catching⟩*
                         ⟨generic op⟩*
   ⟨type definition⟩  ⊨  define ⟨type definition or⟩ [⟨callback⟩]
  ⟨scope definition⟩  ⊨  define scope ⟨scope⟩⟨nlg token⟩ [⟨range⟩]
             ⟨range⟩  ⊨  from ⟨nlg term⟩ to ⟨nlg t⟩ [⟨priority⟩]
          ⟨priority⟩  ⊨  [ run ⟨nlg nt⟩ priority ⟨scope⟩ [⟨callback⟩] ]
          ⟨callback⟩  ⊨   then ⟨callback⟩
⟨type definition or⟩  ⊨  ⟨nlg type⟩⟨nlg token⟩ | ⟨type⟩⟨nlg token⟩
  ⟨type inferencing⟩  ⊨  infer ⟨signature⟩ :  ⟨nlg token⟩ [⟨type inferencing opt⟩]
⟨type inferencing opt⟩ ⊨ => ⟨nlg type⟩ | with [⟨nlg type⟩∗]
     ⟨type checking⟩  ⊨  check ⟨nlg token⟩ :  ⟨nlg type⟩⟨variance⟩⟨nlg type⟩
        ⟨generic op⟩  ⊨  enter scope | exit scope | initRoot
    ⟨error catching⟩  ⊨  try { ⟨program⟩ } [on ⟨nlg exeption⟩ { ⟨program⟩ }]
          ⟨variance⟩  ⊨  covariant | contravariant | invariant
         ⟨nlg token⟩  ⊨  ⟨nlg nt⟩.token
          ⟨nlg type⟩  ⊨  ⟨nlg nt⟩.type
            ⟨nlg nt⟩  ⊨  $⟨digit⟩
             ⟨nlg t⟩  ⊨  #⟨digit⟩
             ⟨digit⟩  ⊨  [0-9]
             ⟨scope⟩  ⊨  the name of the scopes defined in the language
              ⟨type⟩  ⊨  the name of the types defined in the language
         ⟨signature⟩  ⊨  the name of the signatures defined in the language
          ⟨callback⟩  ⊨  the name of the callbacks defined in the language
```

Listing 4.4. *Grammar for the TypeLang DSL..*

In Section 4.2.2, we discussed effective techniques for managing the traversal of an Abstract Syntax Tree (AST) in code. Listing 4.5 provides a concrete example of how the semantic action linked to the production for the declaration of a javascript function can be implemented. The portion enclosed in `0..` corresponds to the semantic action associated with the FunDeclaration node. When the AST in Y is referenced, this code is executed when the sum1 4.2 function is visited.

```
module FunDeclaration {
    imports {
        java.util.concurrent.atomic.AtomicInteger;
    }
    reference syntax {
       // $0              #0         $1         #1  $2      #2  #3  $3             #4
       FunDeclaration <- "function" Identifier "(" FunArgs ")" "{" BlockStatement "}";
       FunArgs <- Identifier "," FunArgs;
       FunArgs <- Identifier;
       FunArgs <- "";

       categories:
            keyword = { "function" },
            brackets = { "{", "}" };

    }

    role(type-checker) <typecheck> {
        0 .{
            var helper = $$CompilationHelper;
            var unit = $$CompilationUnit;

            try {
                eval $1
                // We are creating a new TypeFunction that is a class
                // that extends Type and represent a function
                var type = new TypeFunction();

                //SymbolTableEntryFactory is a factory to create a SymbolTableEntry
                var functionScope = new SymbolTableEntryFactory()
                    .withCompilationHelper(helper)
                    .withCompilationUnit(unit)
                    .withToken($1.token)
                    .withEntryType(type0)
                    // The folding range is enclose between braces "{" "}"
                    .withFoldingRange(Range.foldingRangeFrom($n,3,4))
                    .withEntryKind(EntryKind.DEFINE)
                    //At the end we bind the symbol created to the current scope
                    .bindScopeOrReuse();

                helper.getTaskBuilder()
                    .withContext($ctx)
                    .insideScope(functionScope)
                    .withPriority(Priorities.FUNCTION)
                    .withAstNodes($3)
                    // We create and register a task for node $3
                    .createAndRegisterTask();

                // We execute node $2
                unit.enterScope(functionScope);
                eval $2
                unit.exitScope();
            } catch (NeverlangTypesystemException e) {
                e.submit(helper);
            }
        }.
    }

    role(before-each) {
        0 .{
            AtomicInteger counter = new AtomicInteger(0);
            $ctx.nt(2).streamSymbolList("FunArgs", "Identifier").forEachOrdered(e ->
              e.setValue("pos", counter.getAndIncrement())
            );
        }.
    }

}
```

Listing 4.5. *A Neverlang module that defines a function declaration..*

Key points about this code include the following: Lines 27-39 detail the creation of a new symbol table entry for the function identifier symbol, which is associated with the TypeFunction type; a specific implementation of the Type interface. In lines 31-47, a new task is registered to visit the body of the function when executed. Lines 51-51 involve evaluating the params node within the scope of the newly created function. In line 53, a TypesystemException exception is caught and forwarded on a shared communication channel, confining the error to a specific code region and allowing the AST visit to continue.

Further explanation of the code is typically unnecessary, as programmers usually write Java code using a Domain-Specific Language (DSL) called TypeLang for ease of use. Upcoming sections will provide detailed information on the functionalities of this DSL.

Every terminal symbol can be used as an identifier. In addition to the associated string, the token also contains the position of the identifier, given by the URI of the file and the range of the token. To create a token from a Neverlang nonterminal, use the syntax `Token.fromASTNode(n, 0)`, where 0 represents the index of the nonterminal, corresponding to `#0` in Neverlang-stylish syntax.

The basics of grammar and documentation are outlined in Listing 4.4, which lists all the constructs that make up TypeLang. The provided grammar defines various constructs and operations related to type binding, type scoping, task definition, callback execution, type inferencing, and type checking within a hypothetical language or system. The goal of this grammar is to facilitate the creation and management of types, tasks, and error handling in a structured and automated way.

The first set of rules deals with type binding. By using `define $1.type $2.token`, we establish a new type binding where the content of the `$2` token attribute becomes the identifier, and the content of the `$1` type attribute is the type. Additionally, `define $1 $2` translates to `$1.type $2.token` by default, allowing for more straightforward type binding definitions. The rule `define customType $2.token` optionally adds a token attribute for custom types, indicating flexibility in how types are defined.

Next, the grammar introduces the concept of type scope definition. By defining `scope module $1.token`, a new type scope module is bound to the token of `$1`. This scope is further elaborated by `define scope module $1.token from #0 to #1`, which sets a folding range from `#0` to `#1`. This range likely indicates the span within the source code where this scope is applicable.

In the context of task definition, the rule `define scope module $1.token from #0 to #1` followed by `[ run $1 priority module ]` specifies a new compilation unit task. This task is designed to visit node `$1` with the priority set to `module`. The inclusion of square brackets denotes the encapsulation of the task definition within a specific scope and range.

Callbacks are handled through two primary rules. The first, `define customType $2.token then customCallback`, ensures that a callback function named `customCallback` is executed at the end of binding a custom type. The second rule, `define scope module $1.token from #0 to #1 [ run $1 priority module then customCallback ]`, integrates a callback execution at the end of a task, ensuring custom behavior post-task execution.

Type inferencing rules provide mechanisms to infer types based on specific signatures and tokens. The rule `infer customSignature $1.token` infers a type with a signature named `customSignature` and token `$1.token`. Furthermore, `infer customSignature $1.token => $1.type` stores the inferred type in `$1.type`. The rule `infer function $1.token with [$1 $2]` infers a function type, passing `$1.type` and `$2.type` as parameters to the function constructor signature.

Type checking ensures that types remain consistent. The rule `check $0.token : $0.type is invariant $1.type` verifies that `$0.type` remains invariant when compared to `$1.type`, using `$0.token` for error location purposes. This step is crucial for maintaining type safety and correctness in the system.

Lastly, error catching is handled similarly to the try-catch construct in general-purpose languages (GPLs). The grammar provides a structure for capturing and managing exceptions, ensuring that any caught errors are submitted to `CompilationHelper` for processing. This error management framework is essential for robust and reliable system behavior, allowing for graceful handling of unexpected issues during compilation or execution.

In summary, this grammar provides a comprehensive framework for defining and managing types, scopes, tasks, callbacks, type inferencing, type checking, and error handling. Each production rule contributes to a structured approach to system design, ensuring type safety, task prioritization, and robust error management.

Additionally, TypeLang generates code automatically based on the type system of the target language, ensuring that each DSL adheres to its own type system. For instance, the statement `define customType 12` can be parsed only if the user has associated the keyword customType with a class extending Type. This association is established using the annotations, as documented in Listing 4.6.

We are able to use two different kind of annotations, both of them bring the same result, but the first one is more readable and the second one is more compact.

```java
@TypeAnnotation(TypeEnum.CUSTOM_TYPE)
@TypeLangAnnotation(
    //This is exactly the keyword used in typelang to define this type
    keyword = "customtype",
    //This is the name of the type in typelang
    kind = TypeSystemKind.TYPE
)
public class CustomType implements Type {
    // Other code
}
```

Listing 4.6. *TypeLang annotations..*

Additionally, we have another listing 4.4 showing the difference when using the DSL. This listing, which is about 25 lines of code less, highlights the significantly reduced complexity compared to the previous implementations. The complexity of the earlier code is much higher in comparison to the streamlined approach provided by the DSL.

```
module FunDeclaration {
    imports {
        java.util.concurrent.atomic.AtomicInteger;
    }
    reference syntax {
       FunDeclaration ← "function" Identifier "(" FunArgs ")" "{" BlockStatement "}";
       FunArgs ← Identifier "," FunArgs;
       FunArgs ← Identifier;
       FunArgs ← "";

       categories:
            keyword = { "function" },
            brackets = { "{", "}" };

    }


    role(type-checker) <typecheck> {

        0 .{
            try {
                eval $1
                define scope function $1 from #3 to #4 [ run $2 $3 priority function ]
            }
        }.

        4 @{
            define unresolved $5 position $5.pos
        }.

        7 @{
            define unresolved $8 position $8.pos
        }.
    }

    role(before-each) {
        0 .{
            AtomicInteger counter = new AtomicInteger(0);
            $ctx.nt(2).streamSymbolList("FunArgs", "Identifier").forEachOrdered(e ->
              e.setValue("pos", counter.getAndIncrement())
            );
        }.
    }
}
```

Listing 4.7. *A Neverlang module that defines a function declaration using the DSL..*

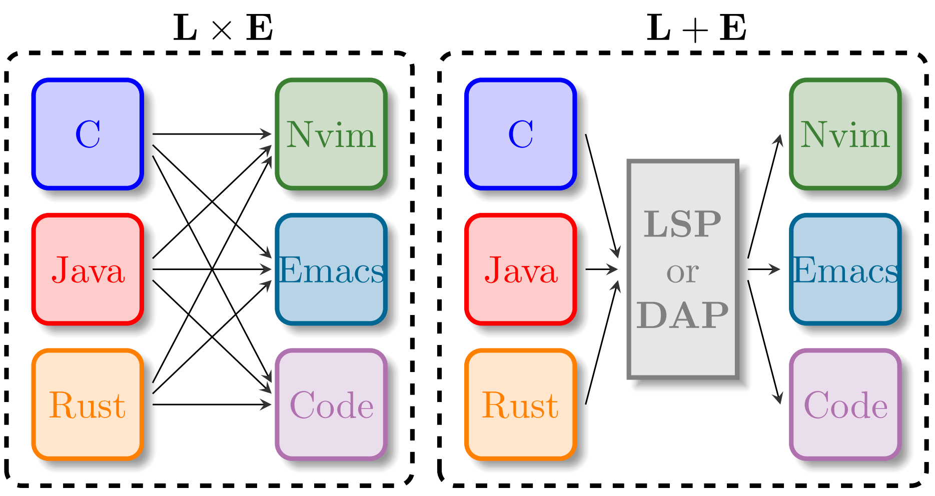


Figure 4.6. *Traditional approach vs LSP/DAP approach to language support..*

4.6 Reduce to L × 1 the number of combinations to support L languages

As shown in Figure 4.6, the traditional approach to language support requires a separate implementation for each language, resulting in a combinatorial explosion of implementations. In contrast, the Language Server Protocol (LSP) and Debug Adapter Protocol (DAP) provide a standardized interface that decouples the language server from the client, allowing a single language server to support multiple languages. This approach significantly reduces the number of required implementations, simplifying the development and maintenance of language servers. In our approach, we aim to further reduce the number of required implementations by introducing a **client generator** that automatically generates client code for a given language bringing the number of required implementations to $\mathcal{L} \times 1$. Currently, there are lots of *editor*, such as *Visual Studio Code*, *Vim/Nvim*, *Emacs* and *IntelliJ IDEA*, that support the LSP and DAP protocols. The client generator will be able to generate the client code for these editors, allowing the language server to be used with any of them.

4.6.1 Syntax Highlighting Generator

One of the key components of a modern code editor is syntax highlighting. Proper syntax highlighting significantly enhances the readability and maintainability of code by visually distinguishing elements such as keywords, operators, strings, and comments. In our approach, we have developed a sophisticated **Syntax Highlighting Generator** that receive in input the keywords and other syntactic elements from a given language specification and generates the corresponding syntax highlighting configuration for various editors.

By parsing the language's grammar and lexical rules, the parser gets keywords, operators, and other syntactic constructs. The generator then maps these elements to appropriate highlighting rules for each supported editor. This process ensures that syntax highlighting is both accurate and consistent across different editors.

Our generator supports a wide range of editors, including Visual Studio Code, Vim/Nvim, Emacs. For each editor, the generator produces a configuration file or plugin that can be easily integrated into the editor's ecosystem. This automation not only saves significant development time but also ensures that the syntax highlighting is always up-to-date with the latest language specifications.

Moreover, the Syntax Highlighting Generator is highly customizable. Developers can specify additional highlighting rules or override the default behavior to accommodate specific needs. This flexibility makes our generator suitable for both general-purpose programming languages and domain-specific languages (DSLs).

In summary, our Syntax Highlighting Generator streamlines the process of creating and maintaining syntax highlighting configurations. By automating this task, we ensure that developers can focus on writing code rather than manually configuring their editors. The result is a more efficient and enjoyable coding experience, regardless of the chosen editor.

4.6.2 LSP Generation

The Language Server Protocol (LSP) has revolutionized the way code editors and IDEs provide language-specific features such as auto-completion, go-to-definition, and refactoring. However, creating an LSP server from scratch for each language can still be a daunting task. To address this challenge, we have developed a comprehensive **LSP Generation** framework that simplifies the creation of LSP servers.

Our LSP Generation framework leverages the language specification to automatically generate the boilerplate code required for an LSP server. This includes setting up the communication protocol, handling initialization, and managing language-specific requests and notifications. By automating these aspects, our framework significantly reduces the amount of manual coding required to implement an LSP server.

The core of our LSP Generation framework is a set of templates and code generation tools that can be customized for different languages. These templates include common LSP server components such as text document synchronization, hover information, and diagnostics. Developers can extend these templates to support additional language features or integrate with existing tools and libraries.

Our framework also includes a comprehensive test suite to ensure the generated LSP server is compliant with the LSP specification and performs reliably. This suite includes unit tests for individual components as well as integration tests that simulate common usage scenarios. By providing a robust testing infrastructure, we help developers deliver high-quality LSP servers with confidence.

One of the key advantages of our LSP Generation framework is its extensibility. Developers can easily add support for new languages or customize existing templates to meet their specific requirements. This flexibility ensures that our framework can evolve with the ever-changing landscape of programming languages and development tools.

Overall, our LSP Generation framework represents a significant advancement in the development of language servers. By automating much of the boilerplate code and providing a robust testing infrastructure, we enable developers to quickly and efficiently create high-quality LSP servers. This reduces the overall development effort and accelerates the adoption of the LSP across different languages and editors.

4.6.3 Client Generator

The heart of our innovative approach lies in the **Client Generator**, a powerful tool designed to automatically generate client code for various editors, thereby reducing the number of required implementations to $\mathbf{L} \times 1$. This remarkable reduction in effort and complexity is achieved through a combination of advanced templating, code analysis, and generation techniques.

1. The Client Generator works by taking a language specification as input and producing the necessary client code for editors that support the LSP and DAP protocols. This process involves several steps:
2. Language Analysis: The generator analyzes the language specification to identify key constructs such as syntax rules, semantic rules, and debugging capabilities. This information is used to tailor the generated client code to the specific needs of the language.
3. Template Customization: Based on the language analysis, the generator customizes a set of pre-defined templates for each supported editor. These templates include code for initializing the client, handling LSP and DAP requests, and integrating with the editor's ecosystem.
4. Code Generation: The customized templates are then used to generate the client code. This code is structured to be easily maintainable and extensible, allowing developers to further customize it if needed.
5. Integration and Testing: The generated client code is integrated with the target editors and subjected to a rigorous testing process. This includes both automated tests and manual verification to ensure the generated code works seamlessly with the language server and provides a smooth user experience.

By automating the generation of client code, our Client Generator dramatically reduces the time and effort required to support new languages in multiple editors. This not only simplifies the development process but also ensures a consistent and high-quality experience for users across different editors.

4.6.4 Modular Reuse and Reduction to $N \times 1$

An even more groundbreaking aspect of our approach is the shift from $\mathbf{L} \times 1$ to $\mathbf{N} \times 1$, where **N** is significantly less than **L**. This incredible reduction is made possible by the modular design of our tools, which allows components developed for one language to be reused across multiple other languages.

This modularity is a game-changer. Components such as syntax highlighting rules, LSP implementations, and debugging configurations can be shared and reused, drastically cutting down the number of unique implementations needed. When a new language is introduced, it can often leverage existing components, requiring only minor modifications or extensions. This reuse is facilitated by the interoperability of our generated modules, ensuring that the core functionalities are consistent and reliable across different languages and editors.

For example, if a new language shares similar syntactic or semantic structures with an existing one, the syntax highlighting and LSP components from the existing language can be quickly adapted and reused. This reuse extends beyond mere code copying; our system is designed to recognize and integrate these components seamlessly, ensuring that updates and improvements to shared components propagate across all languages that use them.

This modular reuse also means that the effort to introduce and support a new language is significantly reduced. Instead of starting from scratch, developers can build on a robust foundation of existing components, focusing their efforts on the unique aspects of the new language. This not only accelerates the development process but also ensures a high level of quality and consistency.

In conclusion, the combination of our Syntax Highlighting Generator, LSP Generation framework, Client Generator, and the modular reuse of components represents a transformative approach to language support. By reducing the number of required implementations from $\mathbf{L} \times 1$ to $\mathbf{N} \times 1$, we achieve unparalleled efficiency and flexibility. This innovative approach allows us to rapidly support new languages, maintain high standards of quality, and provide a seamless and consistent user experience across different editors. Our work is not just a technical achievement but a significant step forward in the evolution of development tools, setting new standards for efficiency and modularity in the software industry.

5 Implementation

All implementations in this chapter will be implemented in the Java programming language.

5.1 The type system implementation

In this section, we describe the implementation of the modular type system described in 4.1 step by step.

5.1.1 Type implementation: the basic building block

```java
public interface Type {

  String id();

  default boolean isAssignableFrom(Type other, Variance variance) {
    return false;
  }

  default boolean matchSignature(Signature signature) {
    return false;
  }

  default Type bind(List<Type> neededTypes) {
    return this;
  }

}
```

Listing 5.1. *The Type interface..*

The Type interface is the basic building block of the type system. As shown in Listing 5.1, the Type interface has four methods:

- `String id()` that returns the unique identifier of the type.
- `boolean isAssignableFrom(Type other, Variance variance)` that returns `true` if the type is assignable from the `other` type with the given `variance`.
- `boolean matchSignature(Signature signature)` that returns `true` if the type matches the given `signature`.
- `Type bind(List<Type> neededTypes)` that returns the type bound to the given `neededTypes`.

The method bind could be used to bind a type to its generic parameters. For instance, the type List<String> is bound to the type List<T> with neededTypes being String. Another usage of the bind method is to bind a type function to its arguments. For instance, the type function concat(T, U) could be bound to the type function concat(String, Integer) with neededTypes being String and Integer.

Note that, the Type interface does not have a method to create a new type. This is because the type system is immutable. Once a type is created, it cannot be changed. This is to ensure that the type system is consistent and that the type system is thread-safe.

Additionally, the Type interface does not have a method to compare two types. This is because the type system is based on the **structural type system** [**?**, **?**]. Two types are equal if they have the same structure. For instance, the type List<String> is equal to the type List<String>. However, note that it is trivial pass to a **nominal type system** [**?**] by adding a method to compare two types based on their unique identifier.

5.1.2 Scope implementation: the context of the type

```java
public interface Scope<IDENTIFIER> extends Type {
  TypingEnvironment<IDENTIFIER> getTypingEnvironment();
  IDENTIFIER identifierFromToken(Token token);
  void setParent(Scope<IDENTIFIER> parent);
  Optional<Scope<IDENTIFIER>> getParent();
  default void applyBinding(IDENTIFIER variable, SymbolTableEntry entry) {
    getTypingEnvironment().bindTypeToIdentifier(variable, entry);
  }
  default Stream<SymbolTableEntry> streamSymbolTableEntries() {
    return getTypingEnvironment().stream()
        .map(Map.Entry::getValue)
        .flatMap(EntryTypeBinder::stream);
  }
  default InferenceResult inferFromSignature(Token token, Signature signature) {
    return new StreamInferenceResult(
      token, signature, streamInternalTypes(identifierFromToken(token), signature)
    ).or(streamExternalVisible(identifierFromToken(token), signature));
  }
  @Override
  default boolean isAssignableFrom(Type other, Variance variance) {
    return switch (variance) {
      case INVARIANT -> this.equals(other);
      case COVARIANT, CONTROVARIANT -> false;
    };
  }
}
```

Listing 5.2. *The* Scope *interface..*

The Scope interface is the context of the type. It is a generic interface that allows to the implementer to define the *Java Type* of the identifier (IDENTIFIER in Line 1 of 5.2).

As shown in Listing 5.2, the Scope interface has seven main methods:

- TypingEnvironment<IDENTIFIER> getTypingEnvironment() that returns the typing environment of the scope.
- IDENTIFIER identifierFromToken(Token token) that returns the identifier from the given token.
- void setParent(Scope<IDENTIFIER> parent) that sets the parent scope of the scope.
- Optional<Scope<IDENTIFIER» getParent() that returns the parent scope of the scope.
- void applyBinding(IDENTIFIER variable, SymbolTableEntry entry) that applies the binding of the given variable to the given symbol table entry.
- Stream<SymbolTableEntry> streamSymbolTableEntries() that returns a stream of the symbol table entries of the scope.
- InferenceResult inferFromSignature(Token token, Signature signature) that infers the type of the given token from the given signature.

The Scope extends the Type interface 5.1.1 to allow the scope to be used as a type. Reinforcing what was said in Section 4.1.2, we would like to keep the scope as generic as possible. This is to allow the scope to be used in different contexts. In addition, every scope has a typing environment that is a map of types (see Section 4.1.5). The identifierFromToken method is used to get the identifier from a token. The setParent method is used to set the parent scope of the scope. The getParent method is used to get the parent scope of the scope. In order to apply the binding of a variable to a SymbolTableEntry 5.1.4, the scope has the applyBinding method. This method is used to bind a variable to a SymbolTableEntry in the scope. The streamSymbolTableEntries method is used to stream the symbol table entries of the scope. The inferFromSignature method is used to infer the type of a token from a signature.

5.1.3 Signature implementation: the definition of a type

```java
public interface Signature {
  default SymbolTableEntry typeResolution(SymbolTableEntry entryType) {
    return entryType;
  }
}
```

Listing 5.3. *The Signature class..*

The Signature class is the definition of a type. As shown in Listing 5.3, the Signature class has one main method:

- `SymbolTableEntry typeResolution(SymbolTableEntry entryType)` that returns the type resolution of the given `entryType`.

As mentioned in Section 4.1.3, in order to perform *type inference*, the `typeResolution` method can be used to do some operations on the type. For instance, the `typeResolution` method could be used to resolve the type of a variable. Another usage of the `typeResolution` method is to resolve the type of a function. For instance, immagine a varbiable that during the **parse** phase its type is marked as *unknown*, because the type could be omitted. During the **resolve** phase, the type of the variable is resolved to a specific type.

5.1.4 Symbol Table Entry Implementation: an entry in the Typing Environment

```java
public interface SymbolTableEntry extends Indexable {
  EntryKind entryKind();
  EntryType entryType();
  EntryDetails details();
  default <T extends Type> T type() {
    return (T) refType().get();
  }
  default AtomicReference<Type> refType() {
    return entryType().refType();
  }
  default Location location() {
    return entryType().token().location();
  }
  default boolean isAssignableFrom(SymbolTableEntry symbolTableEntry,
                                   Variance variance) {
    return type().isAssignableFrom(symbolTableEntry.type(), variance);
  }
}
```

Listing 5.4. *The `SymbolTableEntry` interface..*

`SymbolTableEntry` is used to represent informations about symbols in the typing environment. As shown in Listing 5.1.5, the `SymbolTableEntry` class has three main methods:

- `EntryKind entryKind()` that returns the kind of the entry.
- `<T extends Type> T type()` that returns the type of the entry.
- `Location location()` that returns the location of the entry.

The `entryKind` method is used to get the kind of the entry, the in this case the kind of the entry is an `EntryKind` enum that can assume three different values: `DEFINE`, `USE`, and `IMPORT`. The `type` method is used to get the type of the entry, note that this method retrives the type as a generic type using `entryType` method that return an `EntryType`. An `EntryType` is a class that encapsulates the type of the entry and allow the retrival of the type from a `Token` during the **parse** phase. The `location` method is used to get the location of the entry, the location is a class that encaps the information about the location of the entry in the source code.

Entry Type Binder Implementation: an helper to perform type binding

```java
public interface EntryTypeBinder {

  EntryTypeBinder bindEntry(SymbolTableEntry type);

  boolean isBound();

  Stream<SymbolTableEntry> stream();

  Optional<SymbolTableEntry> getBoundEntry();

  void removeIf(Predicate<SymbolTableEntry> predicate);

}
```

Listing 5.5. *The `EntryTypeBinder` interface..*

The `EntryTypeBinder` class is an helper class that is used to perform type binding. As shown in Listing 5.5, the `EntryTypeBinder` class has two main method:

- `EntryTypeBinder bindEntry(SymbolTableEntry type)` that binds the given `type` to the entry.
- `boolean isBound()` that returns `true` if the entry is bound.

In our idea, a `SymbolTableEntry` should have a `SymbolTableEntry` inside it, that is used to check if the entry is bound or not. And the `bindEntry` method is used to bind the given `type` to the entry.

5.1.5 Typing Environment implementation: the symbol table

The `TypingEnvironment` class is the symbol table of the type system. As shown in Listing 5.8, the `TypingEnvironment` class has three main methods:

- `TypingEnvironment<IDENTIFIER> bindTypeToIdentifier(IDENTIFIER variable, SymbolTableEntry anyType)` that binds the given `variable` to the given `anyType`.
- `Stream<SymbolTableEntry> getTypesBoundedWith(IDENTIFIER t)` that returns a stream of the types bounded with the given `t`.
- `void removeIf(Predicate<SymbolTableEntry> predicate)` that removes the entries that satisfy the given `predicate`.

The `TypingEnvironment` class contains a map (`HashMap<IDENTIFIER, EntryTypeBinder>`) from identifiers to `EntryTypeBinder`. One of the most important concepts is that the `EntryTypeBinder` can change, in fact in the constructor must be passed the class that extends `EntryTypeBinder` interface. This is to allow different kind of binding. Hence, the first method `bindTypeToIdentifier` is used to bind the given `variable` to the given `SymbolTableEntry`. This method could fail if the `variable` is already bounded.

The second method `getTypesBoundedWith` is used to get the types bounded with the given `t`. The third method `removeIf` is used to remove the entries that satisfy the given `predicate`.

5.1.6 Type Inference inference: the ability to infer the type of an expression

```java
public interface InferencingStrategy {
  SymbolTableEntry infer(InferenceResult inferenceResult);
}
```

Listing 5.6. *The `InferencingStrategy` interface..*

The `InferencingStrategy` interface is used to infer the type of an expression. As shown in Listing 5.6, the `InferencingStrategy` interface has one main method:

- `ISymbolTableEntry infer(InferenceResult inferenceResult)` that infers the type of the given `inferenceResult`.

The `InferencingStrategy` interface is used to infer the type of an expression. The `infer` method is used to infer the type of the given `inferenceResult`.

```java
public interface InferenceResult {
  Stream<SymbolTableEntry> stream();

  InferenceResult or(Supplier<InferenceResult> supplier);

  Token token();

  Signature signature();
}
```

Listing 5.7. *The `TypeInference` class..*

The `InferenceResult` class is used to represent the result of the type inference. As shown in Listing 5.7, the `InferenceResult` class has four main methods:

- `Stream<SymbolTableEntry> stream()` that returns a stream of the symbol table entries of the inference result
- `InferenceResult or(Supplier<InferenceResult> supplier)` that returns the result of the given `supplier` if the inference result is empty.
- `Token token()` that returns the token of the inference result.
- `Signature signature()` that returns the signature of the inference result.

5.2 Towards a modular type system implementation

In this section, we describe the implementation of the modular type system described in **??** step by step.

5.2.1 Compilation Unit Implementation: a logical unit of source code

The `CompilationUnit` class is a logical unit of source code. As shown in Listing 5.9, the `CompilationUnit` class has seven main fields:

- `Scope<IDENTIFIER> scope` that is the scope of the compilation unit.
- `AbstractCompilationUnit<IDENTIFIER> compilationUnitParent` that is the parent compilation unit of the compilation unit.
- `Stack<Scope<IDENTIFIER» stack` that is the stack of the scopes of the compilation unit.
- `InferencingStrategy inferencingStrategy` that is the inferencing strategy of the compilation unit.
- `Location location` that is the location of the compilation unit.
- `CompilationUnitTask task` that is the task of the compilation unit.
- `String id` that is the unique identifier of the compilation unit.

Additionally, the `CompilationUnit` class has three main methods:

- `bindTypeToIdentifier(IDENTIFIER typeId, SymbolTableEntry type)` that binds the given `type` to the given `typeId`.
- `InferenceResult typeInference(Token token, Signature signature, Scope<IDENTIFIER> scope)` that infers the type of the given `token` from the given `signature` in the given `scope`.
- `Range foldingRange()` that returns the folding range of the compilation unit.

5.2.2 Compilation Unit Task: a task to perform compilation

In order to perform the compilation, we need a task that is able to perform the compilation. The `CompilationUnitTask` class is used to perform the compilation. As shown in Listing 5.10, the `CompilationUnitTask` class has five main fields:

- `AtomicReference<CompilationUnitToken> token` that is the token of the compilation unit task.
- `Consumer<Context> consumer` that is the consumer of the compilation unit task.
- `Context context` that is the context of the compilation unit task.
- `Class<? extends AbstractCompilationHelper<?, ?» aClass` that is the class of the compilation unit task.
- `PRIORITY priority` that is the priority of the compilation unit task.

Additionally, the `CompilationUnitTask` class has one main method:

- `void run()` that runs the compilation unit task.

5.3 Compilation Helper Implementation: an helper to perform compilation

The `CompilationHelper` class is an helper class that is used to perform the compilation. As shown in Listing 5.11, the `CompilationHelper` class has numerous fields and methods. The methods have been omitted for brevity. Meanwhile, the `CompilationHelper` class has numerous fields:

- `Class<? extends AbstractCompilationUnit<?» compilationUnitClass` that is the class of the compilation unit.
- `Class<? extends LSPGraph> lspGraphClass` that is the class of the LSP graph.
- `Class<? extends SymbolTableEntryFactory<?, ?» symbolTableEntryFactoryClass` that is the class of the symbol table entry factory.
- `Map<String, String> baseTypes` that is the map of the base types.
- `SubmissionPublisher<Object> publisher` that is the publisher of the compilation helper.
- `InferencingStrategy inferencingStrategy` that is the inferencing strategy of the compilation helper.
- `Scope<ID> root` that is the root scope of the compilation helper.
- `AbstractCompilationUnit<ID> rootCompilationUnit` that is the root compilation unit of the compilation helper.
- `CompilationUnitExecutor compilationUnitExecutor` that is the compilation unit executor of the compilation helper.
- `CompilationContext context` that is the context of the compilation helper.
- `SymbolTableEntryFactory<?, ?> symbolTableEntryFactory` that is the symbol table entry factory of the compilation helper.
- `AtomicBoolean rootIsInitialized` that is the flag that indicates if the root is initialized.
- `AtomicInteger incrementalRuns` that is the number of incremental runs.
- `AtomicReference<CompilationUnitToken> lastToken` that is the last token of the compilation helper.
- `AtomicReference<LSPGraph> graph` that is the graph of the compilation helper.

Lots of fields are injected in the constructor of the class, this is to allow the class to be as modular as possible. The `CompilationHelper` class has a lot of methods that are used to perform the compilation.

```java
public class TypingEnvironment<IDENTIFIER> {

  private final HashMap<IDENTIFIER, EntryTypeBinder> map;
  private final Class<? extends EntryTypeBinder> typeBinderClass;

  private TypingEnvironment(HashMap<IDENTIFIER, EntryTypeBinder> map,
                            Class<? extends EntryTypeBinder> typeBinderClass) {
    this.map = map;
    this.typeBinderClass = typeBinderClass;
  }

  public TypingEnvironment<IDENTIFIER> bindTypeToIdentifier(
    IDENTIFIER variable,
    SymbolTableEntry anyType) {
    try {
      var typeBinder = map.getOrDefault(variable, this.typeBinderClass
                                                   .getConstructor()
                                                   .newInstance());
      typeBinder.bindEntry(anyType);
      map.putIfAbsent(variable, typeBinder);
    } catch (InstantiationException | IllegalAccessException
           | InvocationTargetException | NoSuchMethodException e) {
      throw new RuntimeException(e);
    }
    return this;
  }

  public Stream<SymbolTableEntry> getTypesBoundedWith(IDENTIFIER t) {
    if (map.containsKey(t)) { return map.get(t).stream(); }
    else { return Stream.empty(); }
  }

  public void removeIf(Predicate<SymbolTableEntry> predicate) {
    map.entrySet().stream()
      .peek(e -> e.getValue().removeIf(predicate))
      .filter(e -> !e.getValue().isBound())
      .map(Map.Entry::getKey)
      .distinct()
      .toList()
      .forEach(map::remove);
  }

  public Stream<Map.Entry<IDENTIFIER, EntryTypeBinder>> stream() {
    return map.entrySet().stream();
  }

  public static class Builder<IDENTIFIER> {

    private Class<? extends EntryTypeBinder> typeBinderClass =
        MultipleTypeTypeBinder.class;

    public Builder<IDENTIFIER> setTypeBinder(
        Class<? extends EntryTypeBinder> typeBinderClass) {
      this.typeBinderClass = typeBinderClass;
      return this;
    }

    public TypingEnvironment<IDENTIFIER> build() {
      return new TypingEnvironment<>(new HashMap<>(), typeBinderClass);
    }
  }

}
```

Listing 5.8. *The* `TypingEnvironment` *class..*

```java
public abstract class AbstractCompilationUnit<IDENTIFIER> implements Indexable {
  private final Scope<IDENTIFIER> scope;
  private final AbstractCompilationUnit<IDENTIFIER> compilationUnitParent;
  private final Stack<Scope<IDENTIFIER>> stack = new Stack<>();
  private final InferencingStrategy inferencingStrategy;
  private final Location location;
  private CompilationUnitTask task;
  private final String id;

  public AbstractCompilationUnit(
      Scope<IDENTIFIER> scope,
      InferencingStrategy inferencingStrategy,
      AbstractCompilationUnit<IDENTIFIER> compilationUnitParent,
      Location location,
      String id) {
    this.scope = scope;
    this.compilationUnitParent = compilationUnitParent;
    this.inferencingStrategy = inferencingStrategy;
    this.location = location;
    this.id = id == null ? "" : id;
  }

  @Override
  public Range foldingRange() {
    return Optional.ofNullable(location).map(Location::range).orElse(null);
  }

  public void bindTypeToIdentifier(IDENTIFIER typeId,
                                   SymbolTableEntry type)
      throws UnbindableEntryException {
    currentScope().bindTypeToIdentifier(typeId, type);
  }

  public SymbolTableEntry
  bindScopeOrReuse(IDENTIFIER identifier,
                   SymbolTableEntry symbolTableEntry) {
    if (!(symbolTableEntry.type() instanceof Scope<?>)) {
      throw new UnbindableEntryException("Type is not a scope",
                                         symbolTableEntry);
    }
    bindTypeToIdentifier(identifier, symbolTableEntry);
    return symbolTableEntry;
  }

  private InferenceResult typeInference(Token token, Signature signature,
                                        Scope<IDENTIFIER> scope) {
    return scope.inferFromSignature(token, signature)
        .or(()
                -> scope.getParent()
                       .map(parent -> typeInference(token, signature, parent))
                       .orElse(null));
  }

  public void
  updateTaskIfPresent(Function<CompilationUnitTask, CompilationUnitTask> fun) {
    getTask().ifPresentOrElse(e -> setTask(fun.apply(e)), () -> {
      Compiler.logger.warning(
"Something wrong could happen, your root compilation task should not be empty!");
    });
  }

  /* Other methods */
}
```

Listing 5.9. *The* `CompilationUnit` *abstract class..*

```java
public record CompilationUnitTask<PRIORITY extends Comparable<PRIORITY>>(
    AtomicReference<CompilationUnitToken> token,
    Consumer<Context> consumer,
    Context context,
    Class<? extends AbstractCompilationHelper<?, ?>> aClass,
    PRIORITY priority)
    implements Comparable<CompilationUnitTask<PRIORITY>> {
  public CompilationUnitTask(
      Consumer<Context> consumer,
      Context context,
      Class<? extends AbstractCompilationHelper<?, ?>> aClass,
      PRIORITY priority) {
    this(new AtomicReference<>(), consumer, context, aClass, priority);
  }

  public void run(CompilationContext compilationContext) {

    var oldToken = token.getAndSet(compilationContext.token());
    if (oldToken == null) {
      Compiler.logger.fine("Compilation unit run");
      consumer.accept(context);
    } else if (oldToken.equals(compilationContext.token())) {
      Compiler.logger.fine("Already ran");
    } else if (compilationContext.incremental()) {
      Compiler.logger.fine("Incremental run");
      var helper =
          context.root().<AbstractCompilationHelper<?, ?>>getValue(
                  "$" + aClass.getSimpleName()
          );
      compilationContext
          .incrementalCompilationHelper()
          .ifPresent(e ->
              e.beforeCompilationUnitRecompilation(
                  helper, context, compilationContext.token())
              );
      consumer.accept(context);
    }
  }

  @Override
  public int compareTo(CompilationUnitTask<PRIORITY> o) {
    return this.priority().compareTo(o.priority());
  }

}
```

Listing 5.10. *The `CompilationUnitTask` abstract class..*

```java
public abstract class AbstractCompilationHelper<ID, PRIORITY extends Comparable<PRIORITY>>
    implements Flow.Publisher<Object>, AutoCloseable, Submitter<Object> {
  @Inject
  @Named(TypeMapperModule.compilationUnit)
  public Class<? extends AbstractCompilationUnit<?>> compilationUnitClass;

  @Inject
  @Named(TypeMapperModule.lspGraph)
  private Class<? extends LSPGraph> lspGraphClass;

  @Inject
  @Named(TypeMapperModule.symbolTableEntryFactory)
  public Class<? extends SymbolTableEntryFactory<?, ?>> symbolTableEntryFactoryClass;

  @Inject
  @Named(TypeMapperModule.baseTypes)
  private Map<String, String> baseTypes;

  private SubmissionPublisher<Object> publisher = new SubmissionPublisher<>();

  private InferencingStrategy inferencingStrategy = new FindFirstInferencingStrategy();

  private Scope<ID> root;
  private AbstractCompilationUnit<ID> rootCompilationUnit;
  private CompilationUnitExecutor compilationUnitExecutor;
  private CompilationContext context;
  private SymbolTableEntryFactory<?, ?> symbolTableEntryFactory;

  private AtomicBoolean rootIsInitialized = new AtomicBoolean(false);
  private AtomicInteger incrementalRuns = new AtomicInteger(0);
  private AtomicReference<CompilationUnitToken> lastToken = new AtomicReference<>();

  private AtomicReference<LSPGraph> graph = new AtomicReference<>(null);

  /* Other methods */
}
```

Listing 5.11. *The `CompilationHelper` abstract class..*

6
Case Study

6.1 Neverlang

Neverlang is a language workbench that allows the definition of new languages by composing existing languages. It is based on the idea of language-oriented programming, where the language is the main abstraction mechanism. Neverlang is implemented in Java and uses the dexter [?] parser generator to define the syntax of the languages. The semantics of the languages are defined using a combination of Java code and the Neverlang API. The Neverlang API provides a set of abstractions that allow the definition of language constructs, such as statements, expressions, and types. The API also provides mechanisms for defining the behavior of the constructs, such as how they are parsed, type-checked, and executed.

6.1.1 The Neverlang type system

The type system in Neverlang is structured around various priorities that dictate the order of execution for tasks. These priorities are essential for maintaining the proper sequence and dependencies among different components of the language framework.

Priorities

The priorities in Neverlang are used to determine the order in which different components of the language framework are processed. The priorities are defined as a set of constants in the `Priority` class, which is part of the Neverlang API. The priorities are used to ensure that the components are processed in the correct order and that all dependencies are satisfied.

1. **Sources**: This priority is related to the root node, which is the starting point for the execution process.
2. **Module**: After the root node, module declarations are executed. Modules form the fundamental building blocks and need to be processed early.
3. **Syntax**: Syntax declarations are processed next, but they depend on module declarations because syntax can be imported from other modules.
4. **Role**: Roles are linked to the symbols of a production, which means they depend on the syntax.

5. **Slice**: Slices integrate roles and syntax, requiring the completion of both before they can be processed.
6. **Bundle**: Bundles are collections of slices and modules, so they come after both.
7. **Language**: The final priority is given to languages, which depend on the previously processed bundles and slices.

Types

The type system in Neverlang is based on the concept in which the types are used to represent the different constructs in the language framework. These types are categorized based on their functionality within the language framework.

Language Features Related Types:

- **TypeBundle**: Represents a collection of slices and modules.
- **TypeEndemicSlice**: A specific type of slice inherent to a particular context.
- **TypeModule**: Represents a module within the language.
- **TypeSlice**: Represents a slice that ties together roles and syntax.

Internal Module Declaration Types:

- **TypeCategory**: Defines categories within modules.
- **TypeRole**: Represents roles within the module.
- **TypeSyntax**: Represents syntax definitions within the module.

Internal Syntax Declaration Types:

- **TypeNonTerminal**: Non-terminal symbols in the syntax.
- **TypeProduction**: Productions or rules in the syntax.
- **TypeRegex**: Regular expressions used in the syntax.
- **TypeTerminal**: Terminal symbols in the syntax.
- **TypeLanguage**: Represents the overall language construct.

6.1.2 Neverlang LSP

The capabilities that have been implemented are:

- Diagnostics
- Semantic Tokens
- Document Symbols
- References
- Go-to-definition
- Hover information
- Folding ranges

– Inlay hints

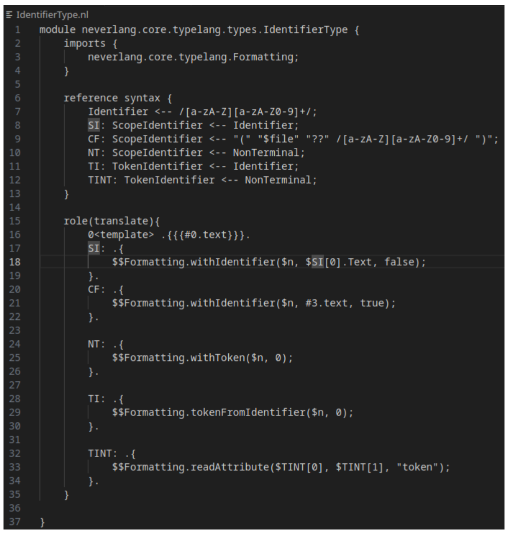
The ability to display symbols identifiers within productions is highly advantageous, as demonstrated in figure ??. The annotations added beside each symbol serve as inlay hints, providing valuable visual cues to the user. The inlay hints are particularly useful for understanding the structure of the code and identifying the different components of the language constructs.

Workspaces are managed in a particular way, in fact the Language Server for each workspace tries to understand if the project is a gradle project:

– if it's a gradle project, all the projects listed in a Gradle file, which may include subprojects, are filtered to only include those with the "neverlang" pllugin. A workspace is created for each of these projects and then:
  – The compilation classpath of the "neverlang" compiler is retrieved, and all classes that implement a "neverlang" unit are decompiled into "neverlang" source code.
  – All the files in the "neverlang" source set, as well as any decompiled files, are analyzed.
– if it's not a gradle project, the Language Server is started for the workspace

Before and After LSP

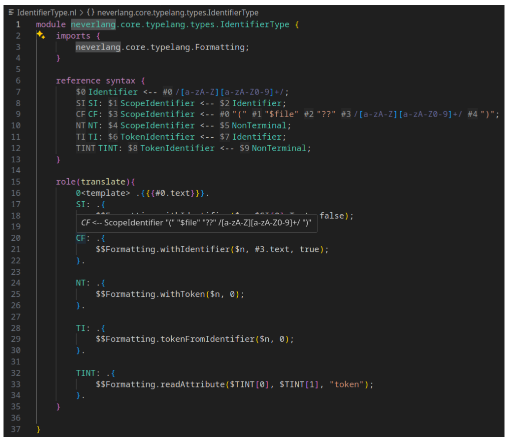
Before the LSP generation, the source code of Neverlang in Visual Studio Code was not syntax-highlighted, and there was no support for go-to-definition or hover information etc. After the LSP generation, the source code of Neverlang in Visual Studio Code is syntax-highlighted, and there is support for go-to-definition, hover information, and other features provided by the LSP.



IdentifierType.nl

```
module neverlang.core.typelang.types.IdentifierType {
    imports {
        neverlang.core.typelang.Formatting;
    }

    reference syntax {
        Identifier <-- /[a-zA-Z][a-zA-Z0-9]+/;
        SI: ScopeIdentifier <-- Identifier;
        CF: ScopeIdentifier <-- "(" "$file" "??" /[a-zA-Z][a-zA-Z0-9]+/ ")";
        NT: ScopeIdentifier <-- NonTerminal;
        TI: TokenIdentifier <-- Identifier;
        TINT: TokenIdentifier <-- NonTerminal;
    }

    role(translate){
        0<template> .{{{#0.text}}}.
        SI: .{
            $$Formatting.withIdentifier($n, $SI[0].Text, false);
        }.
        CF: .{
            $$Formatting.withIdentifier($n, #3.text, true);
        }.

        NT: .{
            $$Formatting.withToken($n, 0);
        }.

        TI: .{
            $$Formatting.tokenFromIdentifier($n, 0);
        }.

        TINT: .{
            $$Formatting.readAttribute($TINT[0], $TINT[1], "token");
        }.
    }

}
```

Figure 6.1. *Neverlang source code before the LSP generation.*



IdentifierType.nl > {} neverlang.core.typelang.types.IdentifierType

```
module neverlang.core.typelang.types.IdentifierType {
    imports {
        neverlang.core.typelang.Formatting;
    }

    reference syntax {
        $0 Identifier <-- #0 /[a-zA-Z][a-zA-Z0-9]+/;
        SI SI: $1 ScopeIdentifier <-- $2 Identifier;
        CF CF: $3 ScopeIdentifier <-- #0 "(" #1 "$file" #2 "??" #3 /[a-zA-Z][a-zA-Z0-9]+/ #4 ")";
        NT NT: $4 ScopeIdentifier <-- $5 NonTerminal;
        TI TI: $6 TokenIdentifier <-- $7 Identifier;
        TINT TINT: $8 TokenIdentifier <-- $9 NonTerminal;
    }

    role(translate){
        0<template> .{{{#0.text}}}.
        SI: .{
            [illegible] false);
        CF <-- ScopeIdentifier "(" "$file" "??" /[a-zA-Z][a-zA-Z0-9]+/ ")"
        CF: .{
            $$Formatting.withIdentifier($n, #3.text, true);
        }.

        NT: .{
            $$Formatting.withToken($n, 0);
        }.

        TI: .{
            $$Formatting.tokenFromIdentifier($n, 0);
        }.

        TINT: .{
            $$Formatting.readAttribute($TINT[0], $TINT[1], "token");
        }.
    }

}
```

Figure 6.2. *Neverlang source code after the LSP generation.*

7 Conclusions