

# Code Less to Code More<sup>†</sup>

## Streamlining Language Server Protocol and Type System Development for Language Families

Federico Bruzzone<sup>✉a</sup>, Walter Cazzola<sup>✉a,\*</sup>, Luca Favalli<sup>✉a</sup>

<sup>a</sup>Università degli Studi di Milano, Computer Science Department, Milan, Italy

### Abstract

Developing editing support for  $\mathcal{L}$  languages in  $\mathcal{E}$  editors is complex and time-consuming. Some languages do not provide dedicated editors, while others offer a single native editor. The *language server protocol* (LSP) reduces the language-editor combinations  $\mathcal{L} \times \mathcal{E}$  to  $\mathcal{L} + \mathcal{E}$ , where a single language server communicates with editors via LSP plugins. However, overlapping implementations of linguistic components remain an issue. Existing language workbenches struggle with modularity, reusability, and leveraging type systems for language server generation. In this work, we propose: i) Typelang, a family of domain-specific languages for modular, composable, and reusable type system implementation, ii) a modular language server generation process, producing servers for languages built in a modular workbench, iii) the variant-oriented programming paradigm and a cross-artifact coordination layer to manage interdependent software variants, and iv) an LSP plugin generator, reducing  $\mathcal{E}$  to 1 by automating plugin creation for multiple editors. To simplify editing support for language families, each language artifact integrates its own Typelang variant, used to generate language servers. This reduces combinations to  $\mathcal{T} \times 1$ , where  $\mathcal{T} = \mathcal{L}$  represents the number of type systems. Further reuse of language artifacts across languages lowers this to  $\mathcal{N} \times 1$ , where  $\mathcal{N} \ll \mathcal{T}$ , representing unique type systems. We implement Typelang in Neverlang, generating language servers for each artifact and LSP plugins for three editors. Empirical evaluation shows a 93.48% reduction in characters needed for type system implementation and 100% automation of LSP plugin generation, significantly lowering effort for editing support in language families, especially when artifacts are reused.

**Keywords:** Software product lines, Feature modularity, Language Server Protocol, Integrated development environments, Software systems architectures, Extensible languages, Domain-specific languages, Neverlang

### 1. Introduction

**Context.** Programming languages require editing support for a proficient use [5]. This applies to both *general-purpose* (GPL) and *domain-specific languages* (DSL). Modern *integrated development environments* (IDE) and *source-code editors*<sup>1</sup> provide editing support capabilities—e.g., highlighting, code completion, and hovering—but the development of such support is complex and time-consuming [121]. Supporting  $\mathcal{L}$  languages across  $\mathcal{E}$  editors requires to implement editing features for each language-editor combination, resulting in a total of  $\mathcal{L} \times \mathcal{E}$  implementations. Language development effort is therefore burdened by the editing support for each targeted editor, whereas editing features can overlap across editors Rask et al. [117]: the risk is introducing inconsistencies and useless implementation overhead. Among many results achieved over the years to improve upon this aspect, we mention type systems implementations [20, 19], *language workbenches* [49] and modular language

development [7, 99, 110, 132]. Bettini [18] demonstrated that type systems are key components for language editing support. For instance, type inference rules can be used to provide inlining hints or error messages and suggestions thereof.

**LSP as a solution.** In 2016, Microsoft, along with RedHat and Codenvy, proposed the LSP<sup>2</sup> [59] as to limit the editing support implementation effort. The LSP model reduces the number of combinations to be implemented from  $\mathcal{L} \times \mathcal{E}$  to  $\mathcal{L} + \mathcal{E}$  by decoupling the implementation of the language editing support from the editor itself. It is based on a component dubbed *language server* which communicates with the editor through an editor-dependent *LSP plugin* (as shown in Fig. 1), both adhering to the LSP specification. The LSP achieves reuse of language services across different editors, but the overlap among different languages remains unaddressed: the number of language servers and the LSP plugins remain  $\mathcal{L}$  and  $\mathcal{E}$  respectively. Type systems development for the language server and LSP plugin generation is complex and usually carried out manually and monolithically in a top-down fashion [18, 19]. Recently, researchers explored *software product lines* (SPL) and their impact on editors [87, 14], which could enhance the modularity and reusability of LSPs and their type systems. SPLs applied to programming languages lead to the creation of *language product lines* (LPL) [86]—i.e.

<sup>†</sup>This work was partly supported by the MUR project “T-LADIES” (PRIN 2020TL3X8X).

\*Corresponding author.

Email addresses: federico.bruzzone@unimi.it (Federico Bruzzone<sup>✉</sup>), cazzola@di.unimi.it (Walter Cazzola<sup>✉</sup>), favalli@di.unimi.it (Luca Favalli<sup>✉</sup>)

<sup>1</sup>For sake of brevity, from here onwards, when not otherwise specified, we will use the term editor meaning both IDE and code editor.

<sup>2</sup><https://microsoft.github.io/language-server-protocol>

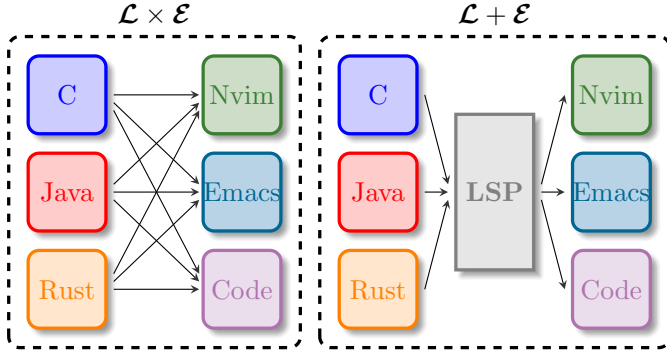


Figure 1: Traditional vs LSP approach to language editing support

SPLs in which each product is a language variant. LPLs have been used in both GPLs [33, 85, 86, 32] and DSLs [34, 61, 93, 133, 140, 141, 50] development.

**Modularization and Reusability Issues.** Notwithstanding this, the editing support is still coupled to the editor ecosystem and can not be reused across different editors—for example, support implemented for Eclipse cannot be reused in IntelliJ IDEA. Most language workbenches are integrated with their native editor and most of them do not provide editing support with modularization in mind. Neverlang [87] is the first language workbench to leverage LPLs towards the reusability of the editing support, yet limited to the Eclipse IDE [87]. Xtext is the only language workbench that generates the language server for the LSP support [24, 25] through XTypeS [15], and its successor XSemantics for the type system definition [17, 18, 20, 19]. Yet, Xtext does not support modular language development and the language server implementation cannot be spread across language artifacts. The main challenge, as Bertolotti et al. [13] reported, lies in the overlap between the implementations of similar linguistic components, which stems from the *monolithic* design of language servers. It hinders their extensibility and reusability in other languages. Maintaining consistency between the language server and the language itself demands time and effort. While  $\mathcal{L} + \mathcal{E}$  is a linear number of combinations, the number of editor  $\mathcal{E}$  still remains significant and cannot be overlooked. As shown in Table 1, none of the analyzed language workbenches provides support for LSP plugin generation, nor for modular language servers.

**Contribution.** In this work, we propose Typelang, a family of domain-specific languages for the implementation of type systems in a modular, composable and reusable way. We demonstrate how the language server implementation can be modularized and reused by automatically generating it for each language artifact with a specific Typelang variant. The Typelang family follows the *variant-oriented programming* paradigm and manages a set of interdependent software components through the *cross-artifact coordination* layer. This approach is particularly suited to complex software systems, which pose numerous challenges [64, 84]. The former defines properties for the integration of product variants within a software system as first-class citizens. The latter provides properties for integrating different variants across various artifacts—self-contained software com-

ponents that may belong to a product line—to address issues of modularity and reusability. Finally, we demonstrate that the proposed approach can reduce the number of combinations required to provide editing support for  $\mathcal{L}$  languages in  $\mathcal{E}$  editors. Initially, the combinations decrease from  $\mathcal{L} + \mathcal{E}$  to  $\mathcal{T} \times 1$ , where  $\mathcal{T}$  represents the set of type systems associated with the languages. This is further reduced to  $\mathcal{N} \times 1$  where  $\mathcal{N} \ll \mathcal{T}^3$  and  $\mathcal{N}$  is the number of type systems without overlaps, achieved by reusing the language artifacts—and their type system—across different languages. The number of editors is reduced to 1 because the LSP client generator can automatically generate the LSP plugin for any editor.

**Evaluation.** To evaluate our approach, we extended Neverlang to support the Typelang family of DSLs and the LSP plugin generation. Using Typelang, we modularly defined the type system for SimpleLanguage<sup>4</sup>—a general-purpose language that is part of the GraalVM project [142]—and Neverlang [132, 135] itself. Then, we generated the plugins for three editors (Visual Studio Code, NeoVim, and Vim). We analyzed the impact of Typelang on the language server generation process, which is entirely based on Typelang: we compared the amount of code needed to implement a type system in Neverlang before and after the introduction of Typelang by calculating the percentage reduction in *lines of code* (LoC) and in the *number of characters* (NoC). We also calculated LoC and NoC needed to develop LSP plugins for the three considered editors to estimate the effort needed to support a new editor. The degree of reusability of type systems across different languages is measured with two metrics: *Normalized Absolute Reuse Degree* and *Operator Conditional Reuse Degree*.

This work is validated by answering the research questions:

- $RQ_1$  To what degree is it possible to streamline by associating variants to language artifacts?
- $RQ_2$  To what degree is it possible to automate the generation of LSP clients, lowering  $\mathcal{E}$  to 1?
- $RQ_3$  Can the language server be automatically generated starting from the Typelang variants lowering  $\mathcal{L}$  to  $\mathcal{N}$ ?

**Structure.** The rest of the paper is organized as follows. Sect. 2 provides foundation, terminology, and background information. Sect. 3 introduces the *variant-oriented programming* paradigm and *cross-artifact coordination* layer, providing a formal definition of the concepts. Sect. 4 presents the Typelang family of DSLs and how it can be used to implement type systems in a modular, composable, and reusable way. Sect. 6 presents the case study and provides the answers to the research questions. Sect. 5 shows how the language server generation can be modularized and reused by automatically generating it for each artifact part of a language variant using its specific Typelang variant. Sect. 7 discusses related work. Finally, Sect. 8 concludes the paper and outlines future work.

<sup>3</sup>The notation  $N \ll T$  means that the number of type systems  $N$  is much smaller than the number of languages  $T$ .

<sup>4</sup><https://github.com/graalvm/simplelanguage>

Language Workbench	Modularity	Separate Compilation	IDE	LS Generation	LS Modularity	LSP Plugin Generation
JustAdd [47]	●	○	○	○	○	○
Melange [45]	⊗	○	3rd p.	★	★	★
MontiCore [58]	●	●	●	○	○	○
MPS [137]	⊗	○	●	★	★	★
Rascal [77]	○	○	●	○	○	○
Spoofax [73]	⊗	●	●	★	★	★
Xtext [16]	○	●	●	●	○	○
Neverlang [135]	⊗	●	3rd p.	★	★	★

Table 1: Comparison of language workbenches in terms of modular development support, separate compilation support, IDE support, language server generation, language server modular development, and LSP plugin generation. The ● symbol indicates full support, ○ no support, ● limited support, ⊗ fine-grained modularization, ⊗ coarse-grained modularization, and ★ indicates no support; however, based on our motivation, it should theoretically be achievable.

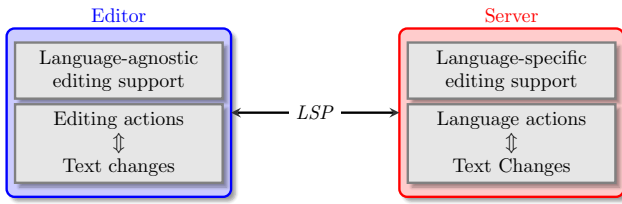


Figure 2: LSP approach to language support. Borrowed from [122]

## 2. Background

We briefly introduce the main concepts and technologies relevant to our work.

### 2.1. Language Server Protocol

In 2016 Microsoft, RedHat and Codenvy defined the LSP. It is a JSON-RPC based protocol that describes the communication between a language server code editor using an LSP plugin. The LSP permits to decouple a language-agnostic editor from the language-specific features, as shown in Fig. 2. The language server provides language-specific analysis, such as go to definition, semantic highlighting, and error checking, while the editor focuses on providing a user-friendly interface and managing the overall development environment. This separation of concerns allows developers to use their preferred editor while benefiting from language-specific features provided by the language server. Consequently, the LSP reduces the number of combinations to implement the language server and editing support from  $\mathcal{L} \times \mathcal{E}$  to  $\mathcal{L} + \mathcal{E}$ , where  $\mathcal{L}$  is the number of languages and  $\mathcal{E}$  is the number of editors (see Fig. 1) [122, 117]. This reduction is achieved by decoupling the implementation of the editing support from the editor. Since its introduction, the LSP has gained significant traction within the development community. Many popular programming languages<sup>5</sup> now have language servers, and various editors support the protocol, making it a *de facto* standard for language support.

The set of commands that a language server can handle is defined by the LSP specification<sup>6</sup>. The specification defines

four categories of commands: *language features*, *text document synchronization*, *workspace features*, and *window features*. The LSP client usually triggers the commands in response to user actions, such as opening a file, typing, or interacting with the editor’s UI. The *language features* respond to requests related to the language semantics, such as code completion, hover information, and go to definition. These requests are sent as a tuple of TextDocument and Position objects, where the TextDocument represents the content of the file being edited, and the Position represents a specific location in the file. The *text document synchronization* category is responsible for keeping the language server up-to-date with the editor’s content via notifications, such as textDocument/didOpen to notify the server that a document has been opened, textDocument/didChange to notify the server that a document has been modified, and textDocument/didClose to notify the server that a document has been closed. The *workspace features* category provides commands to interact with the workspace. A workspace is a collection of files and folders that are opened in the editor. The commands in this category allow the language server to interact with the workspace, such as workspace/symbol to search for symbols in the workspace, and workspace/configuration to retrieve the workspace configuration. The *window features* category provides commands to interact with the editor’s UI, such as window/showMessage to display a message to the user, window/showMessageRequest to display a message with a set of actions, and window/logMessage to log a message to the editor’s console.

### 2.2. Neverlang in a Nutshell

The *Neverlang* [30, 35, 132] language workbench promotes code reusability and separation of concerns in the implementation of programming languages, based on the *language-feature* concept. The basic development unit is the **module**, as shown in line 1 of Listing 1. A module may contain a **reference syntax** and could have zero or multiple **roles**. A role, used to define the semantics, is a composition unit that defines actions that should be executed when some syntax is recognized, as defined by *syntax-directed translation* [1]. Syntax definitions are defined using *Backus-Naur form* (BNF) grammars, represented as sets of *productions* and *terminals*. Syntax definitions and semantic **roles** are tied together using **slices**. Listing 1 shows a simple example of a *Neverlang* module implementing a backup task of

<sup>5</sup><https://microsoft.github.io/language-server-protocol/implementors/servers>

<sup>6</sup><https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification>



```

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 1: Syntax and semantics for the backup task.

the LogLang LPL. Reference syntax is defined in lines 2–6; the *categories* (line 5) are used to generate the syntax highlighting for the IDEs. Semantic actions may be attached to a non-terminal using the production’s label 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. Each **role** is a compilation phase that can be executed in a specific order, as shown in line 24. In contrast, the `BackupSlice` (lines 14–18) reveals how the syntax and semantics are tied together; choosing the **concrete syntax** from the `Backup` module (line 15), and two **roles** from two different modules (lines 16–17). Finally, the **language** can be created by composing multiple **slices** (line 20). The composition in Neverlang is twofold: between modules and between slices. Thus, the grammars are merged 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).

### 2.3. Software and Language Product Lines

Variability-rich software systems development leverages principles from product line engineering, commonly referred to as *feature-oriented programming* [113] and *software product line* [37] engineering. An SPL consists of a family of software products, where their similarities and differences are characterized by their features. A feature is a unit that provides a piece of functionality that satisfies a requirement, represents a design decision, or corresponds to a stakeholder’s interest. A key task in SPL engineering is feature modeling, which involves creating and maintaining a *feature model*. The concept of feature model was first introduced by Kang et al. [71] in the FODA method and serves to represent the variability of a system through its features

and their interdependencies. In SPLs, the feature model formalism is essential for configuring software products by defining valid feature sets, known as *configurations*. A feature is considered *active* if it belongs to the selected subset of features in a configuration, while all other features are deemed *inactive*. The structure of a feature model implicitly captures feature dependencies by specifying mandatory, optional, alternative, and grouped features, alongside parent-child relationships, where a feature can only be *active* if its parent features are also *active*.

On the other hand, the development of families of programming languages and DSLs has gained popularity among researchers and practitioners [106, 88, 40, 143]. Similar to other software, DSL interpreters and compilers can be designed around the concept of product line. 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* [86]. LPLs, widely used [14, 31, 50], have been successfully used in both GPLs [33, 85, 86] and DSLs [61, 134, 133, 141]. Feature-oriented programming [4, 41, 114] embraces the idea of modularizing software systems into feature modules, which encapsulate specific functionality and 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* [75, 76, 90]. Using feature-oriented programming in language development, a family of languages [93] can be defined by composing feature modules [140], and a language can be seen as a product of the family. A special case of a product line is the *multi product line* [123, 124, 125], where multiple product lines are integrated into another software product line.

## 3. Foundational Concepts Overview

As noted by Holl et al. [64], “*managing a set of interdependent software product lines involves numerous challenges*”, e.g. ensuring coexistence among product variants within the same system. Poorly coordinated variant management can increase system complexity and hinder maintainability. To mitigate this, it is essential to adopt programming and architectural approaches that explicitly support per-artifact variant integration, thereby simplifying the process and minimizing the need for source code modifications to either the system or the variants. To this end, we propose the *variant-oriented programming* paradigm and the *cross-artifact coordination* layer as two components that enable the integration of product variants into a complex system.

### 3.1. Variant-Oriented Programming Paradigm

**Overview.** We define the *variant-oriented programming* as a programming paradigm in which product variants are treated as first-class entities within a unique software system, referred to as *variant-oriented software*.

Fig. 3 illustrates, both abstractly and concretely, the *variant-oriented programming* paradigm, in which a (modular) language variant—referred to as *variant-oriented software*—is defined as a subset of artifacts associated with one or more features for the Typelang

and the LSP variant. These variants are defined by the Typelang features and LSP languages, respectively.

This approach aims to improve reusability, modularity, and maintainability of software systems. A *variant-oriented software* is designed to support multiple product variants by providing *shared contexts* that drive the interoperability among variants. Variants interact with the shared contexts to perform their tasks, and the shared contexts can either be global or local. Thus, the paradigm revolves around two concepts: *variants* and *shared contexts*.

In Fig. 3, the *shared contexts* are depicted as the artifact's S.Ctx. and Sym.Tab., which represents its *local contexts*. The *global shared contexts* is illustrated by the gray box at the bottom of the diagram, while the *local shared contexts* are shown as gray boxes that do not overlap the vertical division.

*Variants* are self-contained, independent entities with a feature-based behavior and identity [22], and may also have an internal state. The *Variants* can either be explicitly defined or derived from a set of features, though the latter approach does not guarantee the validity of the derived variants. *Shared contexts* are abstract data types that provide operations enabling the interaction between variants and defining the conditions under which they can interact. These contexts can either have a global scope, making them accessible by all variants, or a local scope, restricting access to a subset of variants.

The dashed lines in Fig. 3 (from the Typelang features to the TC/CI rules) indicate the possibility of deriving the Typelang features from those used in the TC/CI rules. Similarly, the dashed lines from the LSP features to the Typelang features suggest that LSP features can be derived from the corresponding Typelang features.

**Conceptual Example.** Let us take into consideration a *type-checker* product line, which is a family of tools that includes two distinct variants of type checking. For instance, consider a *type checker* with two variants: i) *assignment-statement checker*, and ii) *expression checker*. Both variants traverse their portions of the AST to enforce type safety, each focusing on a distinct node. Now, consider a *compiler pipeline*—representing the *variant-oriented software*—which programmatically<sup>7</sup> selects the *assignment-statement* variant for modules where the correctness of variable bindings is critical, and the *expression* variant for modules where complex expression evaluation requires more thorough analysis. Assuming the AST—serving as the *shared context*—can be checked by both variants, the compiler must be designed to support both strategies and define a clear dispatching policy for assigning each AST node to the appropriate checker. For instance, during the front-end pass the compiler might:

- apply the *assignment-statement checker* to all assignment nodes (ensuring that the left- and right-hand sides are type-compatible);

- apply the *expression checker* to nodes such as BinaryOp, FunctionCall, and other expression constructs (verifying operand types and return types).

This strategy interweaves detection of mis-assigned variables with validation of expression correctness across the AST.

**Integration of Variants.** The *variant-oriented programming* paradigm deliberately leaves the definitions of *variants* and *shared contexts* open. This design choice enables the creation of programming languages that embody the paradigm, referred to as *variant-oriented programming languages*. For example, such a language might automatically handle shared contexts and allow variants to be defined using a syntax like:

```
foo = variant X of Foo ...,
```

where Foo denotes a family of related products.

The Typelang language variant can be defined either explicitly, using a syntax such as

```
foo = variant X of Typelang ...,
```

or implicitly, as illustrated in Fig. 3, through a set of features that uniquely characterize the variant. The same approach applies to the LSP language variant.

The primary goal of this paradigm is to provide strong encapsulation of concerns while promoting interoperability among product variants. By reducing coupling between variants, it enhances the cohesion of the software. At the source code level, *separation of concerns* [69] is achieved through shared contexts, which facilitate the integration of independent variants within a *variant-oriented software*. In *variant-oriented programming*, encapsulation is supported by the design of self-contained variants. These variants can be integrated into a *variant-oriented software* system without requiring changes to the existing code-base or the variants themselves. The paradigm directly addresses the challenge of integrating product variants in complex software systems [43, 108]. When a *variant-oriented software* system serves as a product line, the *variant-oriented programming* paradigm can be seen as an instance of the *multi product line* model [123, 124, 125], enabling interoperability of variants in complex systems.

However, not all product line variants can be seamlessly integrated into a *variant-oriented software*, and not all software systems can support multiple variants of a given product line. Therefore, the concept of *integration* is central to the *variant-oriented programming* paradigm, and certain properties must be satisfied to ensure successful integration.

**Integration Properties.** Given a *variant-oriented software* implementation  $\mathcal{S}$ , a set of *shared contexts*  $C_S = \{\Gamma_1, \Gamma_2, \dots, \Gamma_n\}$  (where each  $\Gamma_i$  is independent), and the variants  $v_i$  and  $v_j$  of a product family  $P$ , we define the *contextual compatibility* relation  $v_i \rightleftharpoons_{\Gamma}^S v_j$ . This relation holds between  $v_i$  and  $v_j$  in  $\mathcal{S}$  wrt. a shared context  $\Gamma \in C_S$  if and only if:

- $v_i$  and  $v_j$  are *independent*;
- they can *coexist* simultaneously in  $\mathcal{S}$ ;
- they are *semantically interoperable* in  $\mathcal{S}$ ; and
- they can *cooperate* on  $\Gamma$  simultaneously.

<sup>7</sup>We assume the *compiler* is configured at setup time rather than at runtime.

A variant-oriented software system must ensure the *coexistence* of variants in  $\mathcal{S}$  during the feature selection phase of the product line engineering process [100]. If two variants cannot coexist, they cannot be integrated into  $\mathcal{S}$  and it should be reported as a constraint violation. Constraints must be explicitly declared so that verification tools can detect such violations. *Independence* of variants must be guaranteed by  $P$  or  $\mathcal{S}$ , especially if  $\mathcal{S}$  generates the variants of  $P$ . When  $P$  is responsible for generating the variants, it could be designed in such a way that the variants are inherently independent or Constraints can be defined to ensure independence. On the other hand, if  $\mathcal{S}$  is responsible for generating the variants, it must enforce independence by design. It could be achieved through design-time analysis (e.g., static verification of configurations) or runtime isolation (e.g., sandboxing, namespace separation). As Briand et al. [23] pointed out for object-oriented systems, *independence* is crucial for maintaining low coupling between objects. Also, the *independence* of variants is essential for safe coexistence in  $\mathcal{S}$ —i.e., if two variants are not independent but must coexist in the same system, they may introduce conflicts (e.g., conflicting dependencies, logical contradictions, or runtime errors). Similarly, *independence* and *coexistence* aim to minimize coupling between variants  $v_i$  and  $v_j$ . *Semantic interoperability* ensures that the variants of  $P$  can be used interchangeably in  $\mathcal{S}$  without any semantic loss and they interact seamlessly, thereby increasing the *cohesion* of  $\mathcal{S}$ . Finally, the *cooperation* of variants on  $\Gamma$  must be determined by both  $\Gamma$  (e.g., enforcing mutual exclusion) and  $\mathcal{S}$  (e.g., preventing deadlocks and race conditions). Thus, the set of variants of a product family  $P$  can be integrated in  $\mathcal{S}$ , denoted as  $P \Rightarrow \mathcal{S}$ , if and only if  $\forall v_i, v_j \in P \mid \exists \Gamma \in C_{\mathcal{S}} : v_i \rightleftharpoons_{\Gamma}^{\mathcal{S}} v_j$ .

In Fig. 3, each artifact provides its own variant of Typelang. These variants are designed to be independent, meaning that they can be used in isolation without relying on other artifacts. For instance, the `Int Type` artifact can function independently of the `String Type` artifact, and vice versa. This independence enables multiple variants to coexist within the same language variant, allowing developers to select only the specific artifacts relevant to their projects. *Semantic interoperability* among variants is ensured by the locality of the artifact's *shared context*, represented by `S.Ctx.` and `Sym.Tab.` For instance, the `Assign Stmt` artifact expects `Sym.Tab.` to be in a specific state during execution. *Semantic interoperability* ensures that switching the Typelang variant used in `Assign Stmt` does not affect the `Sym.Tab.` state. *Cooperation* among variants is supported by the *global shared context*—the gray box at the bottom of the diagram.

### 3.2. Cross-Artifact Coordination Layer

**Overview.** We define the *cross-artifact coordination* layer as a layer that integrates independent product variants into artifacts of an artifact-based *variant-oriented software* without modifying the code of either the software system or the variants. The *cross-artifact coordination* layer has as assumptions all the

properties of the *variant-oriented programming* paradigm. A *variant-oriented software* in the *variant-oriented programming* paradigm definition could not be modular, and it could not have artifacts. The *cross-artifact coordination* layer, built on top of the *variant-oriented programming* paradigm, requires that the software system is artifact-based. This layer helps managing the complexity of integrating product variants across artifacts, particularly when a global shared context allows all product variants in a family to interact with a set of common resources.

Fig. 3 illustrates in the *horizontal dimension* the *cross-artifact coordination* layer, where the Typelang features used by each artifact form the associated variant of the Typelang family, and the same for the LSP features.

**Conceptual Example.** Let us bring back the *type-checker* example to illustrate the concept of *cross-artifact coordination* layer. Suppose that each *module* (the artifact) in the *compiler pipeline*—the artifact-based *variant-oriented software*—has a variant of the *type-checker* product. Each module contains a finite number of *type environments* (the shared contexts) that can be managed by the type-checker variants. Note that type environments can be seen as shared contexts because they are resources that can be accessed by multiple variants, if a module admits multiple type-checking strategies. Now, consider the *global program environment*—the *global shared context*—with several type environments which determine the typing rules of the entire program. The global environment is connected to each module via interfaces. Each type environment is governed by the type checker in the nearest module, which can be either the *assignment-statement checker* or the *expression checker* variant. An issue arises: how can the type-checkers coordinate to enforce consistent typing across module boundaries? To avoid conflicting type conclusions in the global environment, a coordination layer is needed, managing the interaction between the type-checkers and the global type environments. This layer must ensure that the variants can interoperate to maintain a coherent type system across the program.

Fig. 3 illustrates the *LSP Graph* and the *Fenwick Tree* as examples of *global shared contexts*. Each variant populates these structures by adding the features it supports. For example, the *assign statement* artifact adds a node to the *LSP Graph* representing the assigned variable. Additional edges—from variable's uses to its declaration—are added to the *LSP Graph* by other artifacts.

**Integration Properties.** Using the notation from Sect. 3.1, let  $\mathcal{S}$  be an artifact-based *variant-oriented software* and let  $A_{\mathcal{S}} = \{a_1, a_2, \dots, a_n\}$  denote the set of artifacts in  $\mathcal{S}$ . Consider  $P$ , a family of products such that  $P \Rightarrow \mathcal{S}$ . The family  $P$  can leverage the *cross-artifact coordination* layer if and only if, for every  $v_i \in P$  and  $a \in A_{\mathcal{S}}$ , the following conditions are met:

- i)  $a$  is defined in terms of a  $v_i$ ;
- ii)  $a$  provides its specific shared context  $\Gamma \in C_{\mathcal{S}}$  with which  $v_i$  can interact; and
- iii) a *global shared context* of  $\mathcal{S}$ , denoted as  $\Gamma_{\mathcal{S}}$ , must exist to enable semantic interaction among all variants of  $P$ .



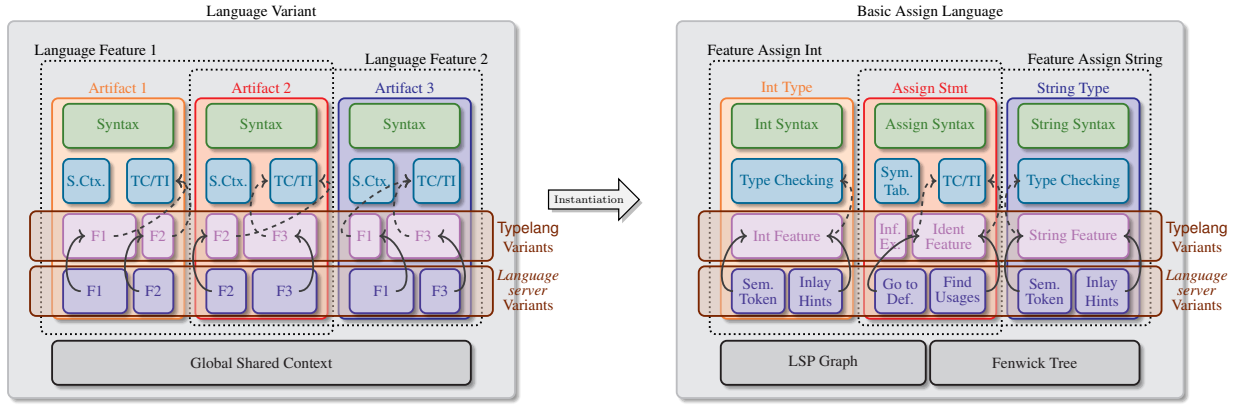


Figure 3: The diagram on the left illustrates the two dimension of variability: (1) *vertical dimension*: each artifact has its own syntax and semantics defined through a grammar and set of rules. The S.Ctx. represents the artifact’s shared context. Additional semantic for type checking (TC) and type inferencing (TI) are specified using the Typelang DSL. The arrows pointing to the TC/CI rules indicate that the Typelang features (F1, F2, etc.) used by the artifact can be inferred from those rules. Finally, the LSP features can also be derived from the Typelang features employed by the artifact. (2) *Horizontal dimension*: this dimension pertains to the *cross-artifact coordination* layer. The Typelang features used by each artifact represent a specific variant within the Typelang family, and the same applies to the corresponding LSP features. The *global shared context* is depicted by the gray box at the bottom of the diagram.

On the right, a concrete instantiation of the abstract diagram is shown to clarify the concept. This example models a simple *assignment* language with three artifact: Int Type, String Type, and the Assign Stmt. The Sym.Tab. is the *shared context* concretely defined by the Assign Stmt artifact.

The first property states that a part of an artifact  $a$ , or its behavior, can be defined by a variant  $v_i$ . Here, a part refers to an essential component of an artifact—one without which the artifact cannot be defined—while behavior denotes the rules governing the artifact’s interaction with other artifacts. The second property states that each artifact  $a$  must maintain a state with which the variant  $v_i$  can interact. This interaction between  $v_i$  and the shared context  $\Gamma$  ensures that the variant has access to the resources it needs to perform its tasks within the artifact. If  $v_i$  does not require interaction with  $a$ ’s state,  $a$  may still be defined by  $v_i$ , though in such cases, the shared context may be empty or entirely omitted. Lastly, the third property is crucial for integrating product variants into artifact-based *variant-oriented software*. It guarantees that all variants within a product family can interact through a global shared context, enabling consistent and coordinated behavior across the system.

Note that, i)  $\mathcal{S}$  can include multiple families of products, each of which can be integrated into  $\mathcal{S}$  using the *cross-artifact coordination* layer, provided the above properties are met, ii) an artifact  $a$  cannot be defined in terms of more than one variant  $v_i$ , iii) the global shared context  $\Gamma_{\mathcal{S}}$  is not simply the union of all shared contexts of the artifacts in  $\mathcal{S}$ , and iv)  $\mathcal{S}$  can provide multiple global shared contexts  $\Gamma_{\mathcal{S}}$ .

In Fig. 3, as previously mentioned, the *type checking* and *type inferencing* behavior of the artifacts is defined by the variants in the horizontal dimension—i.e., the artifacts are defined in terms of the Typelang and LSP variants. The S.Ctx. and the concretely defined Sym.Tab. are the shared contexts of the artifacts, which are defined by the variants in the horizontal dimension. The language variant—the artifact-base *variant-oriented software*—provides two global shared contexts: the LSP Graph and the Fenwick Tree.

### 3.2.1. Considerations

The introduction of *cross-artifact coordination* in artifact-based *variant-oriented software* improves the management of complex product lines. This approach extends variability management from individual artifacts to interactions across multiple artifacts, fostering a comprehensive understanding of modular layers and their relationships across features. It enables seamless integration of variants within a product family and ensures consistent interoperability between artifacts across features and the entire product line. By enhancing modularization, *cross-artifact coordination* layer provides coherence and structure in *variant-oriented software* in spite of the potential high number of feature-artifact combinations.

## 4. Typelang: Towards Type Systems Composition

Bettini [18] demonstrated that type systems are crucial for language editing support. A type system is a set of rules that assigns a type to language constructs, ensuring program’s correctness and the proper applications of operations. Additionally, the type system can be used to infer the type of a construct and verify its compatibility with another construct’s type. The overlap between linguistic components extends to their type systems. In this section, we introduce Typelang, a family of DSLs for defining type systems in a modular, composable, and extensible way. Typelang aims to enhance the reusability of type system definitions by leveraging the *variant-oriented programming* paradigm and a *cross-artifact coordination* layer. Typelang ensures the reusability of type system definitions by coupling Typelang variants to language artifacts.

### 4.1. Overview.

Typelang is a family of DSLs hosted by a language workbench for language syntax and semantics definition. It provides

```

<program>      ::= <type def>* <scope def>* <type infer>*
                | <type checking>* <error catching>*
                | <generic op>*
<type def>     ::= "define" <type def or> [<callback or>]
<scope def>    ::= "define scope" <scope> <lw token>
                [<range>] [<priority>]
<range>        ::= "from" <t> "to" <t>
<priority>     ::= "[" "run" <nt>+ "priority" <scope>
                [<callback or>] "]"
<callback or>  ::= "then" <callback>
<type def or>  ::= <lw type> <lw token>
                | <type> <lw token>
<type infer>   ::= "infer" <signature> <lw token>
                [<type infer opt>]
<type infer opt> ::= <type infer spec>
                | "with" "[" <lw type>+ "]"
                [<type infer spec>]
<type infer spec> ::= ">" <lw type>
<type checking> ::= "check" <lw token> ":"
                <lw type> "is" <variance> <lw type>
<generic op>   ::= "enter" <scope>
                | "exit" <scope>
                | "init" <scope>
<error catching> ::= "try" "{" <program> "}"
                ["on" <exception> "{" <program> "}"]
<variance>     ::= "covariant"
                | "contravariant"
                | "invariant"

```

(a) core module

```

<lw token>     ::= token associated to a <t> or <nt>
<lw type>      ::= type associated to a <t> or <nt>
<nt>           ::= non-terminal defined by the LW
<t>            ::= terminal defined by the LW
<exception>    ::= exception defined in a modular way
<scope>        ::= scopes defined in a modular way
<type>         ::= types defined in a modular way
<signature>    ::= signatures defined in a modular way
<callback>     ::= callbacks defined in a modular way

```

(b) mutable hooks

Listing 2: EBNF Grammar for Typelang

rules for i) type definition, ii) type checking, iii) type inferencing, and iv) error catching.

Its goal is to decouple type system definitions from a specific technological space, enabling artifacts to be used across different language workbenches. A specific Typelang variant can define the type system for a whole language variant or of a number of its language artifacts. Therefore, multiple variants of Typelang can be used to target different language artifacts within the same language workbench to define their type systems. The following example illustrates a snippet of Typelang code.

```

infer identifier <id> with [<expr>.type] => <id>.type
check <id> : <id>.type is invariant <expr>.type

```

It is used to infer the type to an identifier <id> with respect to the expression <expr>. Then, it checks if the inferred type is invariant with the type of the expression. In Typelang, the grammar (Listing 2) consists of an immutable core providing essential functionalities and mutable hooks that allow adaptation to language-specific requirements. By integrating language-specific features with the immutable core, Typelang can be configured into a tailored variant within its product line.

### Typelang as a Family of Domain-Specific Languages.

Typelang is best understood not as a single DSL, but as a *family* of DSLs, organized according to a *language product line* architecture. Its design enables the systematic derivation of language variants for different domains. For example, in some domain-specific languages, the ability to declare functions may be unnecessary. In such cases, the corresponding Typelang variant can be instantiated without the function declaration feature. Conversely, certain application domains may require features that were not originally anticipated during the design of Typelang. The modular nature of the language allows for the extension of the feature set to accommodate such unforeseen requirements.

Modeling Typelang as a family of DSLs rather than a single monolithic DSL serves two primary purposes:

- language restriction**—it enables the creation of minimal, domain-specific variants by omitting unnecessary language features, thereby simplifying the language surface;
- language extension**—it supports the introduction of new, potentially unconventional or unforeseen features, enhancing Typelang’s ability to express a wide variety of type systems.

#### 4.1.1. Core Module.

The fixed part of the grammar (Listing 2(a)) is designed to support language server generation. In Typelang, the entry point of the DSL is `<program>`, which consists of a sequence of statements. New type bindings are defined using the **"define"** keyword, which must be followed by two elements: a type—either a `<lw type>` or a `<type>`—and a `<lw token>` to associate the type with a token. An optional `<callback>` can be specified using the **"then"** keyword, allowing the definition of a callback function that is triggered when the respective binding occurs.

**Type Inference.** Type inference rules in Typelang are defined using the **"infer"** keyword, which depends on the presence of `<signature>`s. The **"infer"** keyword is followed by a `<signature>` and a `<lw token>` to associate the inferred type with a token. The `<signature>`s are designed to support both *structural type systems* [28, 38] and *nominal type systems* [39, 111]. If the type is known in advance, the programmer can help the inference process specifying the type through the **"=>"** operator, followed by a `<lw type>` to indicate the type to be inferred. If the type constructor is not nullary<sup>8</sup>, the **"with"** keyword can be used to specify its argument types. A type constructor can represent various types, including a *product type*, a *sum type*, a *function type*, or a *parametric polymorphic type*. As an example, Fig. 4 layer 4 presents the Typelang code used to define the type inference rules for the assignment statement defined in layer 2. Likewise, Section 6 shows our implementation of Typelang with the type inference rule for the assignment statement in Neverlang. For example, the following snippet defines a type inference rule for a hypothetical assignment statement that infers the type of the identifier <id> from the type of the expression <expr>.

<sup>8</sup>A nullary type constructor is one that does not take any arguments.



```
try {
  infer identifier <id>.token with [<expr>.type] => <id>.type
}
```

**Type Checking.** Type checking in Typelang is defined by using the **"check"** keyword. This keyword is followed by a **<lw token>** and two types, separated by the variance type. The **<lw token>** associates the type checking rule with a token. The developer can explicitly define the variance type between the two types using the **"covariant"**, **"contravariant"**, or **"invariant"** keywords. Similar to type inference rules (shown in boxes denoted with TC/TI in Fig 3), Fig. 4 layer ④ presents the Typelang code defining the type checking rules for the assignment statement, while Section 6 showcases our Neverlang implementation. For example, the following snippet defines a type checking rule for a hypothetical assignment statement that ensures the left-hand side and the right-hand side share the same type.

```
check <id>.token : <id>.type is invariant <expr>.type
```

**Scope Management.** As Cooper and Torczon [39] explained, scopes are crucial for defining the visibility and lifetime of bindings. Typelang enables the programmer to define scopes at the artifact level using the **"define scope"** keyword. The **<scope>** represents the name of the scope, and the **<lw token>** associates the scope with a token. Two elements can be specified: **<range>** and **<priority>**. The former uses the **"from"** and **"to"** keywords followed by two terminals (e.g., the { and } tokens) to set the folding range for the language server. The latter specifies the order in which scopes are executed; each **<scope>** automatically defines a priority level, but the programmer can specify the scope's priority within square brackets using the **"run"** keyword, followed by a sequence of non-terminals, the priority level, and an optional **<callback>**. This configuration indicates that all the non-terminals should be executed with the specified priority level. Note that the language workbench must provide a way for the programmer to define a total order between priority levels. The **"enter"** and **"exit"** keywords, followed by the **<scope>**, are used to define the scope's entry and exit points. Typically, this involves pushing and popping the scope from a stack. For example, the following snippet defines a scope named **module** with a priority level with the same name, which is executed upon entering the module scope

```
define scope module <mod> [
  run <mod> priority module
]
```

**Error Handling.** Some operations may fail during the type checking and type inference phases. The **"try"** and **"on"** keywords are used to define error-handling rules. If an error occurs during the **"try"** block, the **"on"** block is executed. If the **"on"** block is not specified, the error is automatically caught and passed to the *compilation helper* (see Sect. 4.2).

**Root of the Scope Hierarchy.** Finally, the **"init"** keyword is used to define the root of the scope hierarchy tree, followed by the **<scope>** to specify the primary scope. Note that if

the **<scope>** used in the **"init"** keyword should have the highest priority level because it is the first scope to be executed during the type checking and type inference phases.

#### 4.1.2. Mutable Hooks.

The variable part of the grammar (Listing 2(b)) entails the concepts of **<scope>**, **<type>**, **<signature>**, and **<callback>**. For example, an implementation of the **int** type can be hooked to **<type>** to create a Typelang variant with definition, inference, and integers type checking capabilities. This enables two main reuse opportunities: i) the resulting Typelang variant can be leveraged to define multiple type systems involving similar types; and ii) The **int** type can be reused in different Typelang configurations. The concrete implementations of **<scope>**, **<signature>**, **<type>**, and **<callback>** hooks can be developed independently, allowing them to be reused across various Typelang variants.

**Typelang Configuration.** Configuring a Typelang DSL and implementing Typelang-based type systems primarily focus on the **<scope>** and **<type>** concepts. For instance, if a language artifact defines two scopes (*function* and *class*) and two types (*int* and *float*), then the corresponding Typelang variant grammar will include the following definitions:

```
<scope> ::= "function" | "class"
<type>  ::= "int" | "float"
```

Such types are then involved in the definition of **<signature>**s, to provide a way to specify the expected types of an expression. The **<signature>**s hook can be used to decorate untyped parse trees or abstract syntax trees with type definitions and inference rules. For example, consider the function call **let res = add(1, 2);** in Rust, where *add* is a *parametric polymorphic* function over the type **T**, defined as:

```
fn add<T>: std::ops::Add<Output = T>>(a: T, b: T) -> T { a + b }
```

Although the monomorphization phase has not yet been performed,<sup>9</sup> the **<signature>** **add: (T, T) -> T** can be used to infer the type of the **res** identifier based on the argument types of the function call.

The type system can be further refined through the **<callback>** hook, where each callback serves as a unique identifier for a function invoked in response to a specific event. Each of these core concepts is abstract, and the corresponding Typelang rule is flexible, enabling programmers to define their own **<scope>**, **<type>**, **<signature>**, and **<callback>** elements. The semantics of the variable parts of Typelang variants are not predetermined; their implementation is left to the language workbench. In Listing 2(b), such cases are indicated by the phrase *"defined by the language workbench"*, meaning the left-hand side of the production rule is fixed, while the right-hand side must be provided by the language workbench.<sup>10</sup> For example, **<t>**

<sup>9</sup>Monomorphization typically occurs after the type checking and inference phases.

<sup>10</sup>We assume that the fixed part of the Typelang grammar is in a language workbench-compatible format; otherwise, it must be adapted accordingly.

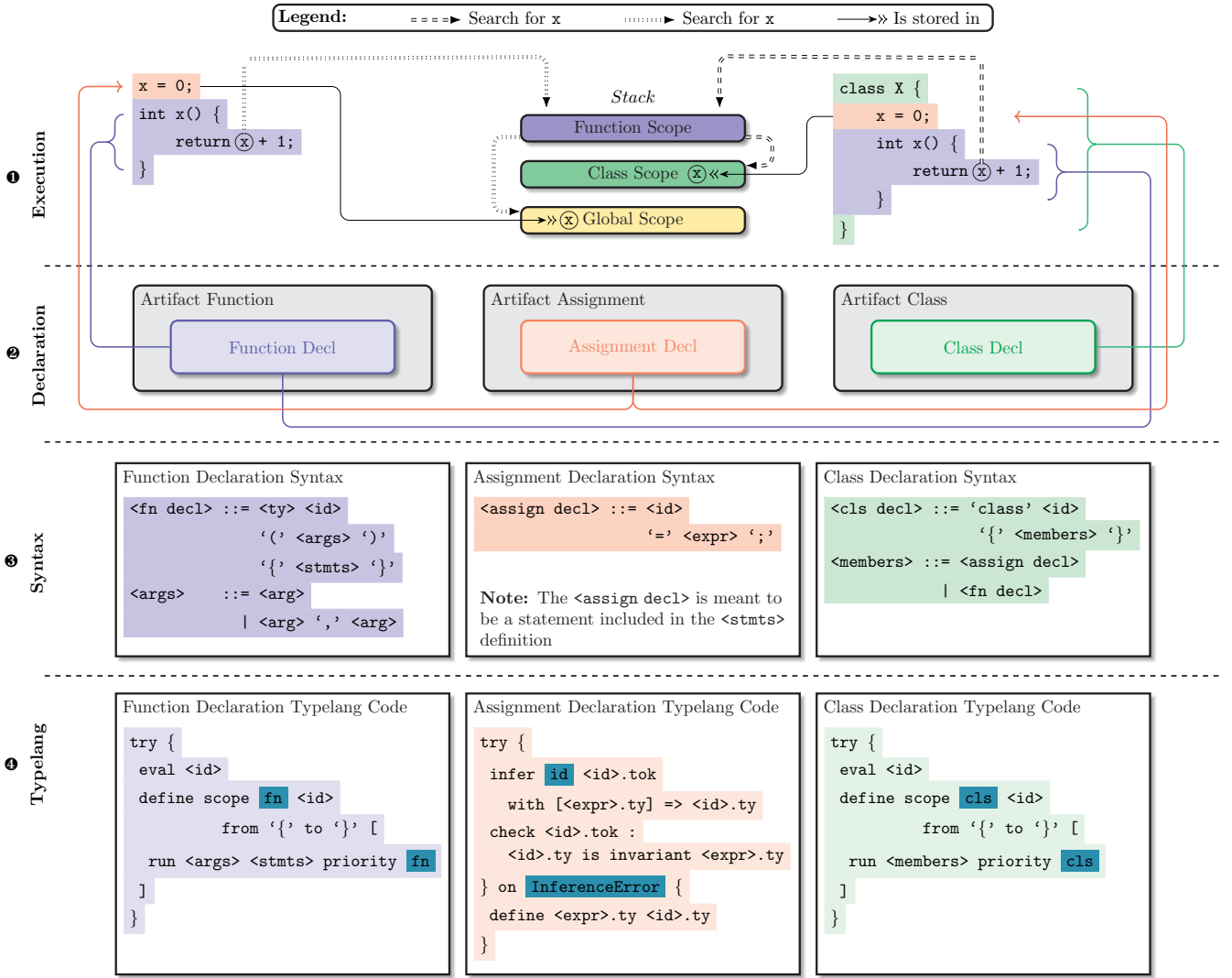


Figure 4: The diagram shows the four layers of Typelang in action. **Execution Layer 1:** Divided into three vertical sections. The left shows a *function scope* created within the *global scope*; the right shows a *function scope* within a *class scope*. The center displays the contents of each scope and their hierarchical relationships. **Declaration Layer 2:** Includes three artifacts—*function*, *class*, and *assignment*. Each artifact contains the same information shown in Figure 3. Scopes are defined modularly, and the grammar supports nesting. **Syntax Layer 3:** Defines the artifacts' syntax modularly. From this grammar, the language workbench generates the *hooks* used in the Typelang DSL. **Typelang Layer 4:** Contains the Typelang code defining type checking and inference rules for the artifacts (denoted in Fig. 3 by TC/TI boxes). Typelang features—called *hooks*—are highlighted in teal.

and `<nt>` serve as placeholders for terminal and non-terminal symbols associated with syntax tree nodes, as defined by the language workbench. Similarly, `<lw type>` and `<lw token>` fulfill analogous roles, referencing syntax tree nodes. However, this distinction highlights whether nodes are accessed directly or used to retrieve their type (`<lw type>`) or source code (`<lw token>`). The `<exception>` construct is designed to be modular and can be raised during the type checking and inference phases. The language server can catch these exceptions and provide feedback to the editor.

#### 4.2. Collection and Assembling Phases

**Overview.** Inspired by the *application engineering* phase of *product line engineering* [112, 94], the Typelang DSL relies on: the *collection* phase and the *assembling* phase. During the

*collection* phase, the language workbench gathers all features specified through *product configurations*. These configurations can be explicitly defined by the programmer or implicitly derived from the semantics of the language artifacts, following the principles of the *variant-oriented programming* paradigm. This phase determines how the Typelang variant should be generated. The *assembling* phase combines the concepts of *product derivation* [115, 44, 65, 57, 116] and *product validation*. In this phase, the language workbench creates the Typelang variant for a given language artifact based on the features collected during the *collection* phase. By assembling the Typelang variant before the compilation phase, the necessary language features are made available during the compilation of the language artifacts.

**Collection Phase.** In our proposal, the *collection* phase gathers all definitions to generate the Typelang variant for a

given language artifact. The variant generation is demanded to the *assembling* phase. Any definition that should be used during *assembling* phase but not collected by the *collection* phase will cause the *assembling* phase to fail. Therefore, the Typelang DSL grammar is not fixed: it depends on the language artifact requirements and it is defined before the artifact is processed. We define a *feature box* as a concrete implementation of a feature. A feature box contains all the necessary information to define the variable part of the Typelang DSL grammar. A feature box combines concepts from *feature modules* [9, 72] in feature-oriented programming and *delta modules* in delta-oriented programming [42, 80]. From feature-oriented programming, we inherit the concept of a *feature module* as a modular unit that encapsulates a feature definition [3]. From delta-oriented programming, we inherit the concept of a *delta module* as a unit that specifies changes to be applied to the core module to implement further products by adding, modifying, and removing code [127].

**Feature Box.** Each variable definition must be encapsulated within its own feature box, such as a language workbench module or a Java class. Feature boxes should be independent but composable when needed. A feature box acts as a provider of a definition and contains all relevant LSP information associated with that definition. For example, a feature box can define a `<type>` and specify whether it serves as an LSP *semantic token*—a token enriched with semantic information used to provide language-aware editor features such as semantic highlighting and code navigation. A concrete example of a feature box is shown in Listing 4. To facilitate development, language workbenches should provide default feature boxes for common definitions while allowing developers to define new ones modularly. This approach suggests that an increasing number of feature boxes will likely result in more Typelang variants.<sup>11</sup> All feature boxes are designed to be reusable across multiple language artifacts to form distinct Typelang variants. These variants can be applied across different languages and, by extension, across various language variants. However, a feature box may depend on—or rather, require—the existence of another feature box to work correctly. Each feature box can be viewed as a provider of a definition, and dependencies between them can be considered *requirements* necessary to utilize the definitions provided by other feature boxes. This brings forth to the need for a *composer*. The *composer*, e.g., a language workbench language compilation unit, is responsible for collecting all the necessary feature boxes to generate Typelang variants.

**Assembling Phase.** The *composer* is responsible for distributing all necessary information to the internal components of the language workbench to support the type checking and type inferencing compilation phases using a specific *assembled* Typelang variant. A proper Typelang variant can be automatically generated for each language artifact, as the *collection* phase solely collects the features actually used in the semantics related to type checking and type inferencing rules. This approach avoids the need for an explicit, verbose form—such as

```

1 public interface SymbolTableEntry extends Indexable {
2     EntryKind entryKind();
3     EntryType entryType();
4     EntryDetails details();
5
6     default Location location() {
7         return entryType().token().location();
8     }
9
10    default <T extends Type> T type() {
11        return (T)refType().get();
12    }
13
14    default AtomicReference<Type> refType() {
15        return entryType().refType();
16    }
17
18    default boolean isAssignableFrom(
19        SymbolTableEntry ste, Variance variance) {
20        return type().isAssignableFrom(ste.type(), variance);
21    }
22 }

```

Listing 3: The SymbolTableEntry interface.

`ty_1 = variant {F1, F2, ...} of ty`—favouring a desugaring mechanism that generates the variant based on the features used in the semantics of the language artifact. With this approach, the *product configuration*, *product derivation*, and *product validation* phases of the *application engineering* phase can be automated. Figure 3 shows the *assembling* phase for a given language artifact (the dashed gray arrows). For instance, the type checking semantics of Artifact 1 uses the Typelang feature F1, while the type inferencing semantics relies on feature F2. The *composer* collects the feature boxes required by F1 and F2 to generate the appropriate Typelang variant for Artifact 1. The teal-colored layer (Fig. 4 ④) highlights the Typelang features used by different artifacts, introduced as *hooks* in the Typelang grammar. Indeed, the other Typelang code shown in Fig. 4 is the *core* of the Typelang variant. Each variant only includes the features needed to define the type checking and type inference rules for the corresponding artifact and remains unaware of features used by other artifacts. The type checking and type inference semantic actions assume that the required features are present in the Typelang LPL and that the *composer* can successfully collect them. A number of issues may arise during the *collection* and *assembling* phases. For instance, if a feature box declaration—such as `fn`, `id`, and `cls` in layer ④ of Fig. 4, or the identifier used in Listing 7—is not found during the *collection* phase, it should be treated as a parsing error.

#### 4.3. Typelang Integration in Modular language workbenches

Using the notation introduced in Sect. 3, let the *variant-oriented software*  $S$  represent a modular language workbench, and let  $P$  denote the Typelang LPL. The *shared contexts*  $\Gamma_i \in C_S$  are the *typing environments*. A *typing environment* is a map from identifiers to symbol table entries, which contain information such as the identifier type and additional metadata (details in Sect. 5). As shown in Listing 3, a symbol table entry is a data structure that contains information about the type associated to the entry, the location in the source code, and other relevant information.

<sup>11</sup>Note that adding new feature boxes may not necessarily lead to new variants.



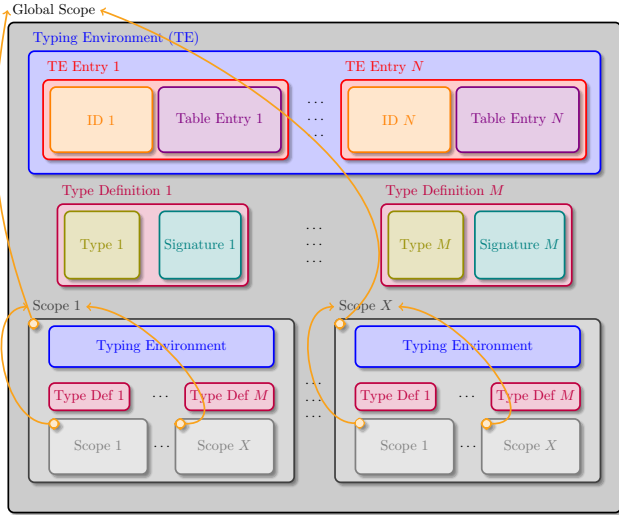


Figure 5: The modularity and extensibility of Typelang, illustrating the independent definition and composition of scopes, types, and signatures as reusable building blocks that can be assembled into language-specific variants at compile-time. The diagram demonstrates a simplified version of Typelang scopes structure, with each scope associated with a typing environment that contains the unordered pairs of identifiers and table entries and types.

**Variant-oriented Programming.** The *composer* generates Typelang variants in accordance with the principles of *variant-oriented programming*. It enforces the first property, that is, that each variant is generated for its language artifact irrespective of the other variant generations. Moreover, each variant is also unaware of the other variants existence. The *coexistence* property is satisfied because  $\mathcal{S}$ , as a modular language workbench, perceives  $v_i$  and  $v_j$  as different configurations of the same DSL, which are already used in modularized language artifacts. The language workbench enforces constraints on the Typelang variants through internal APIs, which serve as an abstraction layer to interact with the variants. The language workbench does not need to be aware of the specific Typelang variant used in a given language artifact; instead, it only needs to know how to interface with the variants and enable their cooperation within the same shared context  $\Gamma$ . As a result, all Typelang variants are considered *semantically interoperable* within  $\mathcal{S}$ , as they can be used interchangeably. The definition of a scope involves the definition of a *typing environment*  $\Gamma$ . It is the responsibility of  $\mathcal{S}$  to ensure correct access to  $\Gamma$  by the Typelang variants. This may require mechanisms such as mutual exclusion to regulate access, handling race conditions to prevent data corruption, and addressing potential issues related to data consistency.

**Modular Scopes.** Fig. 4, layer ④, illustrates the Typelang code, where the *artifact function* and *artifact class* define their own scopes using distinct Typelang variants. The eligibility for scope nesting is directly derived from the grammar (layer ③) defined at the artifact level. In this example, both the *artifact class* and the *global scope* grammars permit function scope definitions. Consequently, the relationships between scopes do not need to be predetermined; instead, they can be established at compile time based on the grammar of the language artifact used

in the language variant. Similarly, the *assignment declaration* is valid within all scopes, including the *global scope*, *function scope*, and *class scope*. In the left-side of layer ①, the variable  $x$  is declared within the *global scope*, whereas in the right-side, it is declared within the *class scope*. When  $x$  is used, its value is retrieved from the nearest enclosing *nested lexical scope*.<sup>12</sup> In the first case, the value is retrieved from the *global scope*, while in the second case, it is retrieved from the *class scope*. According to Cooper and Torczon [39], the *execution layer* can be entirely generated at compile time based on the Typelang scopes defined within the artifact, even if the scopes are modularly defined. Proper data structures that dynamically track the most recent binding of a free variable<sup>13</sup> must be generated. Being the artifacts self-contained ensures that all required feature boxes are included and foster their reuse across different language implementations. Fig. 5 expands and abstracts the stack shown in the center of Fig. 4, illustrating the content of scopes and their hierarchical relationships. This basic representation could be further detailed with elements such as *callbacks* and *exceptions*, although this is beyond the scope of this work.

**Cross-artifact Coordination.** Once Typelang variants are generated and the type systems are defined, the language workbench can use them like a conventional type system for type checking and type inference phases. Scopes in Typelang are defined at the artifact level, each providing multiple type definitions along with an associated *typing environment*  $\Gamma$ . A type definition consists of the type itself and its associated signature. With blue boxes shows in Fig. 5,  $\Gamma$  is represented as a map linking identifiers to symbol table entries. The global scope typing environment is initialized after the invocation of "*init*" `<scope>`, so that the context  $\Gamma$  and type definitions are defined at the artifact level. Each artifact  $a \in A_{\mathcal{S}}$  provides its own interoperability context  $\Gamma \in C_{\mathcal{S}}$ , where variant  $v_i$  can operate, thus satisfying the first property of the *cross-artifact coordination* layer. As shown by Fig. 4, layer ②, it is possible for an artifact  $a$  to omit scope definitions. The omission of scope definitions occur for several reasons:

- a language does not require scopes—e.g., a simple calculator language;
- not all artifacts want to define namespaces—e.g., an assignment statement;
- an artifact depends, after the *composition* phase, on the scopes defined by other artifacts—e.g., a function call.

Moreover, each artifact  $a$  can be defined in terms of a variant  $v_i$ , given that its type checking and type inferencing semantics are derived from  $v_i$ . Consequently, as long as the behaviors of  $a$  are defined with Typelang, the artifacts are indirectly defined by the Typelang variants. Although not depicted in Fig. 5, the global shared context  $\Gamma_{\mathcal{S}}$  acts as a repository for common resources accessible to all Typelang variants for seamless interaction. This context could be implemented as a priority queue containing *compilation unit tasks* (further details in Sec.5), which are exe-

<sup>12</sup>According to Cooper and Torczon [39], “scopes that nest in the order they are encountered in the program are called lexical scopes”.

<sup>13</sup>A *free variable* is a name defined outside the current scope.

```

1 package neverlang.core.defaults.types;
3 /* imports */
5 @TypeAnnotation(Type.FN)
6 public class TypeFunction extends AbstractTypeFunction {
8     @Override
9     @DocumentSymbol
10    public SymbolKind documentSymbol(SymbolTableEntry entry) {
11        return super.documentSymbol(entry);
12    }
14    @Override
15    @SemanticToken({SemanticTokenTypes.Function,
16                    SemanticTokenTypes.Operator})
17    public String semanticToken(SymbolTableEntry entry) {
18        return super.semanticToken(entry);
19    }
21    @Override
22    @InlayHint
23    public InlayHint inlayHint(SymbolTableEntry entry) {
24        return super.inlayHint(entry);
25    }
27    /* semantic methods */
28 }

```

Listing 4: The TypeFunction feature box.

cuted lazily and in increasing priority order. Variants interact by adding their tasks to the queue using the "run" keyword.

## 5. Generative Modular LSP Design

**Overview.** The LSP defines a common API for language servers, enabling a single implementation to be used across multiple editors (clients). It operates on a *client-server* model, communicating over *pipes* or *sockets*. Language server implementations are created manually in a *top-down* fashion, with significant time invested in complex data structures and algorithms [59]. In contrast, we propose a *bottom-up* [85], modular approach to generate language servers. This method leverages the variant-oriented programming and cross-artifact coordination principles. LSP capabilities are conceptualized as separate *feature boxes*, composed at the artifact level for a more flexible and reusable implementation.

**LSP Features and Feature Boxes.** The features of the language server variant—represented as solid gray arrows in Fig. 3—are derived from the Typelang feature. Since a Typelang feature box could require multiple LSP features, the language server variant is composed of the union of the features provided by the feature boxes used by the artifact. Thus, the language server variant is tailored to the artifact’s needs, without unnecessary features and *feature boxes* contain the necessary information to provide LSP capabilities. The language server implementation emerges through a generative process involving the definition of feature boxes, their Typelang representations, and their composition. Fully modular feature boxes—*i.e.*, without endogenous bindings—extend their modularization and reusability to the LSP capability implementations, so that type system components can be reused across languages. Finally, this approach (Fig. 2) takes advantage of the LSP’s language-agnostic nature [107, 121]. Since the protocol imposes no restrictions

on language server implementation, each LSP implementation needs to comply with the protocol specification, with no concern for the other implementations.

### 5.1. Language Server Generation Process

In the LSP context, the implementation of a language server can be seen as a *generative* process based on Typelang feature boxes. As explained in Sect. 4, the feature boxes encapsulate all the needed information to generate the language server implementation.

**Feature Box Definition.** For example, Listing 4 shows the TypeFunction feature box which implements the LSP capabilities for *semantic tokens*, *document symbols*, and *inlay hints*. In our implementation, the *collector* mechanism is realized as an annotation processor. Specifically, the @TypeAnnotation annotation and the value Type.FN (line 5) tell the annotation processor to bind the TypeFunction Java class to a feature box of kind function type. The Type.FN identifier in the annotation corresponds to the kind of feature box and becomes part of the variable section of Typelang’s grammar whenever the feature box is employed. The @DocumentSymbol (line 9) and @InlayHint (line 22) annotations mark specific methods, enabling the client to collect document symbols and provide inlay hints for the function type. At line 15, the @SemanticToken annotation is followed by a list of SemanticTokenTypes—a class provided by LSP4J—that specifies the token types, allowing the client to semantically highlight source code based on these types. Overall, Listing 4 consolidates all LSP capabilities related to the function type without assumptions on other language features: the feature box is then reusable across any language with functions, regardless of its broader structure.

**Feature Box Collection.** All annotated classes are collected only if the feature box associated with the specified kind is actually used by Typelang in a semantic action. Consequently, LSP-related annotations are processed only when the corresponding feature box is employed in the artifact, and the code for the respective language server features is generated only under these conditions. This process is repeated for all feature boxes used by Typelang within an artifact and subsequently for all artifacts associated with a language implemented in the language workbench. The union of all capabilities specified by the feature boxes constitutes the active LSP features for the language server variant of the artifact. Other modular language workbenches can implement the *collector* mechanism and similar methods using alternative strategies, such as visitor patterns [54].

### 5.2. Building Blocks for the Language Server

A language server implementation requires in-memory runtime data structures to properly work, providing all the necessary information to the LSP client. The language workbench is in charge of providing the necessary data structures to generate the language server implementation. Among these data structures, some are automatically populated by Typelang during the execution of the tasks, while others are still automatically populated, but are used by the language server to provide correct responses

to the LSP client requests. This section provides an overview of the data structures that the language workbench must provide to generate the language server implementation.

**Table Entry.** The table entry is at the core of compiler design and type systems. Out interface for the *Table Entry* is shown in Listing 3. It must contain at least the following fields to provide the information to the LSP client:

- the *entry type*, which represents a symbol—ideally a lexical token—in the source code;
- the *entry kind*, which represents the kind of the usage of the symbol in the source code—e.g., a definition or a reference to a symbol;
- the *entry location*, which represents the location of the symbol in the source code;
- the *entry range*, which represents the range of the symbol in the source code.

Some feature box provided capabilities need a *Table Entry* to compute a query and to generate the language server implementation (see Listing 4 at line 10, 15 and 23). In addition to the fields, the implementation includes routines for computing information required by the LSP client. For example, the *isAssignableFrom* routine determines whether the right-hand side of an assignment is compatible with the left-hand side, enabling the client to deliver precise diagnostics. Further, behind the method invocation at line 24 in Listing 4, the *getSignature* routine is called to provide the signature of the symbol used in the inlay hint. Hovering capabilities are provided by the *getHover* routine, which is used to provide relevant information upon hovering over a symbol in the code. The LSP provides a wide range of capabilities that can be implemented by the language server, for more details see the documentation for either the LSP or LSP4J. All the pieces of information regarding table entries are collected within the *typing environment* (shown in Fig. 5); the typing environment is a data structure that maps the relevant symbols in the source code to their respective table entry. The language workbench generates the code to populate the table entry at runtime. Thus, the language server only needs to access the data structure and provide the information to the client.

**Compilation Unit.** For each scope defined in Typelang, a *Compilation Unit* is created and associated to a *Compilation Unit Task*. The compilation unit is a data structure that contains:

- the *scope*, which contains the typing environment  $\Gamma$  associated with it;
- a reference to the *stack* of scopes shown in Fig. 4;
- the *type inference strategy* [54], used by the language workbench to infer the types of the expressions in the scope;
- the associated *compilation unit task* with the associated priority of the task;
- the *parent compilation unit*, which is used to ensure the tasks are executed in the correct order.

The language workbench generates the code to populate the compilation unit at runtime. The compilation unit serves as a container for the tasks that are executed to bring the compilation unit to a fully typed state via the type inference. Without a fully typed compilation unit, the language server cannot provide

correct information to the client. Combining compilation units and incremental parsers with error recovery [2, 55, 119], a small modification made to the source code triggers the execution of only the tasks related to the compilation unit and its scope, as well as any of its dependencies. Other tasks do not need to be repeated.

**Compilation Unit Task.** The *compilation unit tasks* are needed to solve the following limitation: in any given moment, a compilation unit may not be fully typed—e.g., because the type of a symbol is not yet defined or was not inferred yet. Such an example is shown in Fig. 6: the sum identifier in Task #1 (on the left side) with priority fun is highlighted in red because it is not typed yet. The language server cannot provide correct information to the client if the compilation unit is not fully typed. The compilation unit task component contains the following fields:

- the reference to the *language workbench context*, used to access a mutable reference of the AST;
- a procedure,<sup>14</sup> which accept a *language workbench context* and performs some operations on it, such as modifying the AST;
- the *priority* of the task.

As discussed in Sect. 4, the priority is used to order the tasks in the language workbench. To ensure modularity, the feature boxes are unaware of the existence of any other priorities except their own. However, the language workbench and the language server must have a global knowledge of all existing priorities to order the tasks correctly, thus bringing the compilation unit to a fully typed state. This dependency should be declared exogenously, using a language construct that specifies the evaluation order of tasks without affecting the individual feature boxes. This solution embraces the *Hollywood principle* [131, 52] design pattern<sup>15</sup> guaranteeing that the separation of concerns and the reusability are preserved. To provide a visual representation of the tasks and their priorities, Fig. 6 presents an AST with associated *compilation unit tasks* denoted as boxes around the AST nodes. By assuming a left-to-right depth-first AST traversal, a task associated with the global scope is added to  $\Gamma_S$  with the highest priority. The function *sum1* is processed before the function *sum*. Since the body of *sum1* calls *sum* with the formal parameter *x* and an integer literal 1 as actual parameters, this may result in an incomplete AST and related errors. Each variant  $v_i$  is responsible for creating and adding its corresponding *compilation unit task* to the priority queue  $\Gamma_S$  with the appropriate priority level as detailed in Sect. 5.

**Compilation Unit Executor.** The *Compilation Unit Executor* is a component that is used by the language workbench to execute the tasks in the correct order. It contains:

- the *execution listener*, which is used to listen to the execution of the tasks and report the errors to the *Compilation Helper* (see the last paragraph);

<sup>14</sup>A procedure is a function that does not return a value.

<sup>15</sup>“Don’t call us, we’ll call you” is what the Hollywood Principle is all about—known also as Hollywood’s Law or Inversion of Control. It is a design pattern that allows low-level components to hook themselves into a system, but the high-level components determine when they are needed, and how.



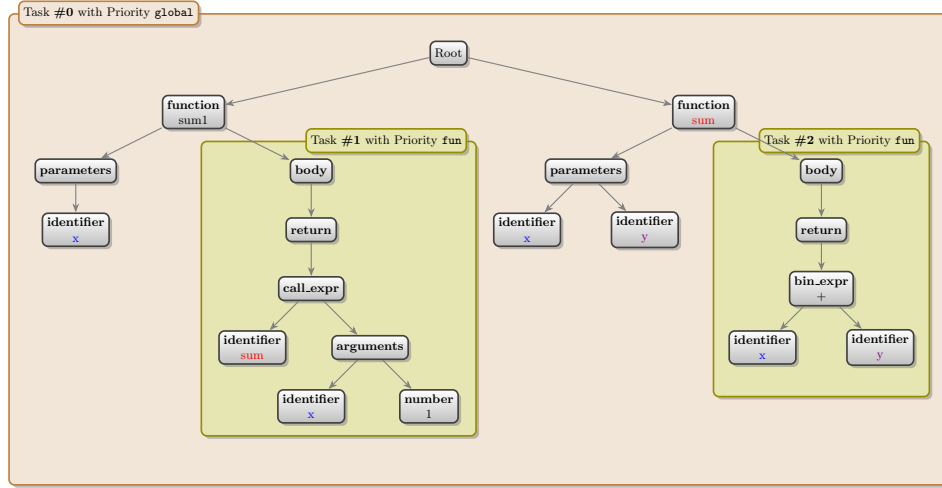


Figure 6: The diagram shows an AST with the *compilation unit tasks* associated.

- the *executor*, which accepts a configuration and executes the tasks concurrently according to the *executor service*;
- the *executor service*, which provides routines to manage termination and methods that can produce a *Future* for tracking the progress of asynchronous tasks;
- the *task queue*—a min-heap priority queue—in which the task dependencies are lazily registered and resolved during the execution of the tasks;
- the *task provider* which is used to provide the tasks to the *task queue*.

The compilation unit executor is used to run the tasks in the correct order, and it is essential for the language server to bring the compilation units to a fully typed state. Concurrency and asynchronous issues could arise during task execution, and the compilation unit executor is in charge of managing these issues.

**Fenwick Tree.** The *Fenwick Tree* [51, 126]—also known as binary indexed tree—is a data structure that provides search and insert operations in  $O(\log n)$  time complexity. In our approach, we decided to use the Fenwick tree to represent the lexical scopes of the source code. Basically, the Fenwick tree is a *global shared context*  $\Gamma_S$  and it is paramount to the language server to provide responses to the LSP client efficiently. The language server leverages the Fenwick tree to provide the *folding range* capability to the client and to efficiently perform scope-based search activities. During the editing, the client can remove or add new scopes, and the Fenwick tree must be updated accordingly. Note that the removal of a scope causes the removal of one subtree of the Fenwick tree. Just as all other data structures, the Fenwick tree is populated automatically by the language workbench during the tasks execution.

**LSP Graph.** The *LSP Graph* is the second *global shared context*  $\Gamma'_S$  used by the language server. It is a symbol dependency graph [78]. Its nodes represent the symbols within the source code and its edges represent dependencies between symbols. A symbol can be a type, a function, a variable, etc. The dependencies between the symbols are based on the usage and

the kind of the symbol in the source code. However, as soon as the name resolution phase is completed, the language workbench starts to populate the LSP Graph with the dependencies between source code symbols. The LSP Graph is used by the language server to provide the *FindReferences*, *GoToDefinition*, and *CodeCompletion* features to the client. It is important to note that the LSP Graph is populated automatically by the language workbench during the tasks execution. Without it, many LSP features would not be available. Note that the edges are not necessarily symmetric. This means that if a symbol A depends on a symbol B, it does not mean that B depends on A. The LSP Graph is used by the language server to provide the information to the client efficiently.

**Compilation Helper.** Data structures face challenges due to the exogenous approach to modularity and a lack of awareness of other components. Data related to the creation of these data structures is provided directly via *Typelang* or indirectly via the language workbench. The *Compilation Helper*, introduced in this approach, aims at providing a centralized place to manage the data structures and the tasks execution, ensuring that the data structures are populated correctly and that the tasks are executed in the correct order. Thus, the compilation helper represents the external component declaring the bindings between all the aforementioned (independent) components—*i.e.*:

- the *root* of the compilation units;
- the *compilation unit executor*;
- the *Fenwick tree*;
- the *LSP Graph*.

The language server uses the compilation helper to interact with the LSP graph and to queue any tasks. A compilation helper reference is passed to the language server variants to allow variants to interact with the data structures and with the tasks execution. Additionally, the compilation helper collects error events reported by the execution listener. These may include *Typelang* exceptions or injected errors detected in the generated code by the language workbench. The compilation helper is responsible for making error management decisions. For example, if the

error is related to the type inference, the compilation helper can decide to re-execute the tasks in the compilation unit, achieving a fully typed state. As a centralized place in which all the errors are managed, the compilation helper can either choose i) to handle the errors as compilation errors—e.g., stop the compilation and emit the errors in standard error/output—or ii) to update the data structures, handling the errors as LSP errors—e.g., keeping alive the language server and emit the errors to the client. Note that, the *Compilation Helper* is used to manage both the type system errors and reusing the same error handling mechanism for the language server errors.

Typelang variants are essential to the language server, both for the errors it provides and for the data structures it populates. Each artifact derives its language server variant from the Typelang variant used to implement the language. The *collecting* and *assembling* phases, as well as type definitions, are inherited from those of Typelang. A language server variant is composed by the features provided by the Typelang feature boxes, used by the respective artifacts. This means that each artifact is unaware of all the features offered by the LSP, knowing only those that are actually needed by the artifact itself. An artifact with its feature boxes can be shipped to another language variant to correctly generate the language server variant. The reusability is guaranteed by the exogenous feature boxes definition approach: LSP capability implementations and type system components can therefore be reused across different languages.

### 5.3. Language Server Integration in Modular language workbenches

By recalling the properties presented in Sect. 3, as already done for Typelang (Sect. 4), we outline how the language server variants can benefit of *variant-oriented programming* paradigm and *cross-artifact coordination* layer principles when integrated within a language workbench. As introduced in Sect. 3, let  $\mathcal{S}$  be a modular language workbench (our *variant-oriented software*), and  $P$  be the language server SPL.  $P \Rightarrow \mathcal{S}$  holds if and only if the four principles of *variant-oriented programming* are observed, meaning that the language server variants are in  $\Rightarrow_{\Gamma}^{\mathcal{S}}$ -relationship.

**Variant-Oriented Programming.** The LSP features used to generate the respective language server variant  $v_i$  are directly provided by each feature box used by the associated Typelang variant: for any  $v_i, v_j \in P$ ,  $v_i$  and  $v_j$  are orthogonal to each other, as they are generated by different feature boxes. There is no restriction imposed by the *variant-oriented programming* paradigm on the generation of variants which depend on another variant, as long as the  $\Rightarrow_{\Gamma}^{\mathcal{S}}$ -relationship is preserved. Therefore, the *independence* principle of variant-oriented programming is upheld. Unlike Typelang, the granularity of the feature activation is at the feature box level, and the language server variant is composed by the features provided by the Typelang feature boxes used by an artifact. This denotes that the simultaneous *coexistence* of  $v_i$  and  $v_j$  in  $\mathcal{S}$  is guaranteed by construction. As explained in Sect. 5.1,  $\mathcal{S}$  provides the necessary data structures—often automatically populated—to the language server variants, providing the information to the LSP client.  $v_i$  and  $v_j$  must be

```

1  LSPClient {
2      generatorVersion = "1.0.1-SNAPSHOT"
3      clientImplementations = [
4          "x-client",
5          "y-client",
6          "z-client",
7          /* ... */
8      ]
9
10     templateGeneratorClasses = [
11         "x.XTemplateGenerator",
12         "y.YTemplateGenerator",
13         "z.ZTemplateGenerator",
14         /* ... */
15     ]
16
17     languageName = "exlang.ExLang"
18     launcher = "lsp.PipeLauncher"
19     fileExt = "ex"
20     binPath = "..."
21 }

```

Listing 5: Gradle build file to generate LSP plugins and the syntax highlighting.

able to dialogue with each other to provide correct information to the LSP client. The *semantic interoperability* property is respected; it is a *must-have* property for the language server variants to be able to co-exist in  $\mathcal{S}$ . The *variant-oriented programming* requires the existence of a shared context  $\Gamma \in C_{\mathcal{S}}$  where  $v_i$  and  $v_j$  can co-operate. Lots of shared contexts  $\Gamma$  are provided by the language workbench, such as the *Fenwick tree* and the *LSP graph*. The *co-operation* is also achieved at runtime, where the language server variants use the compilation helper to dynamically interact with these shared contexts  $\Gamma$ .

**Cross-Artifact Coordination.** Each artifact  $a \in A_{\mathcal{S}}$  can be defined in terms of the language server variant  $v_i$ . Or better, if  $a$  wants to provide the LSP support for its feature boxes, it must allow itself to be defined in terms of the language server variant  $v_i$ . The shared context provided by each artifact are two-fold: the *typing environment*  $\Gamma$ —inherited from the Typelang variant—and the *compilation helper*  $\Gamma'$ . Each  $a$  provide the *typing environment*  $\Gamma$  which is populated by the Typelang variant and it is used by the language server variants to provide the information to the LSP client. The compilation helper is the shared context  $\Gamma'$  that each artifact  $a$  provides to the language server variants to retrieve information already computed related to the LSP capabilities. Two global shared contexts are provided by the language workbench to the language server variants: the *Fenwick tree*  $\Gamma_{\mathcal{S}}$  and the *LSP graph*  $\Gamma'_{\mathcal{S}}$  explained in Sect. 5.1.  $\Gamma_{\mathcal{S}}$  and  $\Gamma'_{\mathcal{S}}$  are populated and interrogated by the language server variants to provide the information to the LSP client. By clarifying these aspects, the *cross-artifact modularization* layer for the language server is achieved.

### 5.4. LSP Plugin and Syntax Highlighting Generation

Improvements in the editing support are also possible by leveraging the language workbench capabilities. In general, since the language workbenches know the syntax and semantics of the language, generating the LSP plugin and syntax highlighting leads to a reduction in the effort required to provide this support. To provide a complete editing support, the editors

```

1 module stmt.IfStatement {
2   imports { /* ... */ }
3
4   reference syntax {
5     /* if syntax */
6
7     categories:
8       keyword = { "if", "else" };
9   }
10
11   /* if semantics */
12 }

```

Listing 6: Example of a Neverlang module that provides the implementation of the *IfStatement* artifact with *if* and *else* categories made available

usually require the implementation of LSP plugins for the language server—which is responsible for providing all the LSP capabilities to the client—and the *syntax highlighting* support. As shown by recent work, Neverlang provides the *categories* introduced by Kühn et al. [87] that have been used to generate the syntax highlighting for LPL as a third-party plugin for the Eclipse IDE [105]. In the same work, the authors proposed the generation of the semantic support for the IDEs, such as the *Semantic highlighting* and *Debugging* features by leveraging the language workbench capabilities. Furthermore, Monticore [81], Rascal [130] and Spoofox [74] provide the editing support for the Eclipse IDE. For more details refer to Table 1.

**Overcoming the IDE Dependency.** To overcome the direct dependency to the Eclipse IDE, thanks also to the advent of the LSP and the widely used TextMate [56, 128]<sup>16</sup> grammar, we extend the proposed approach to provide a methodology to generate the syntax highlighting and the LSP plugin for all the editors that support the LSP. Developer side, a Gradle plugin generates the necessary files for the syntax highlighting and the LSP plugin. This aims to further reduce the effort towards this support and increase the reusability of the language workbenches. Listing 5 shows the Gradle build file that generates the LSP plugin and the syntax highlighting. Developers should only specify the fields shown in Table 2 inside of `LSPClient` block.

Name	Line	Description
generatorVersion	2	The version of the plugin
clientImplementations	3	The list of the desired editors
templateGeneratorsClasses	10	The list of the associated template generators
languageName	17	The name of the language
launcher	18	The language server launcher
fileExt	19	The extension of the language
binPath	20	An executable file containing the language server

Table 2: Fields to specify in the `LSPClient` block of the Gradle build file to generate the LSP plugin and the syntax highlighting.

It is important to note the editor list and the associated template generators can be extended by the developers to support more editors. The support is implemented for the widely used VS-Code [46], NeoVim, and Vim editors, including both the syntax

highlighting and the LSP plugin. Sect. 6 will show the implementation effort in terms of LoC and the NoC. With this approach, we embrace the opportunity offered by language workbenches to provide the syntax highlighting elements. For instance, the Neverlang’s *categories* construct as shown in Listing 6 and the Spoofox’s ability to provide the syntax highlighting elements for NWL as shown by Kats and Visser [73].

**Syntax Highlighting Generation.** Listing 6 shows how the *IfStatement* artifact defines the *if* statement using the terminal symbols *if* and *else*. In the *categories* block (line 7), it exports the keyword category (line 8), which includes the terminal symbols *if* and *else*. Two cases can be distinguished: 1) the syntax element are provided by the already modularized boxes and the language workbench can collect them and perform their union without duplicates (e.g., the Neverlang language workbench), and 2) the syntax elements are provided by the language constructs (e.g., the Spoofox language workbench). The Gradle plugin merely needs the syntax elements to be provided, so that the template generators can use them to generate the syntax highlighting editor support. The Gradle plugin requires the programmer to specify, among other fields, the `binPath`—i.e., the path to the jar file containing the implementation of the language server. The server is then launched by the editors according to the `launcher` field upon opening a file with the extension specified in the `fileExt` field. Additionally, template generators can be written in any JVM-compatible language, as the Gradle plugin uses reflection to instantiate each template generator and invoke the methods that conform to the contract defined by the plugin interface.

## 6. Demonstration Case Study

We demonstrate the applicability of our approach through a case study implemented using the Neverlang language workbench. We chose Neverlang due to its compatibility with the flexibility, extensibility and modularity requirements of variant-oriented programming. The implementation consists of approximately 13,000 LoC (approximately 370,000 NoC) of Java code and approximately 2,000 LoC (approximately 60,000 NoC) of Neverlang code including *Typelang* code.

To assess our approach, we address the following research questions:

*RQ<sub>1</sub>* To what degree is it possible to streamline by associating variants to language artifacts?

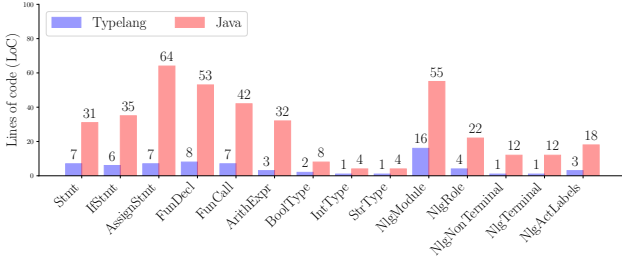
*RQ<sub>2</sub>* To what degree is it possible to automate the generation of LSP clients, lowering  $\mathcal{E}$  to 1?

*RQ<sub>3</sub>* Can the language server be automatically generated starting from the *Typelang* variants lowering  $\mathcal{L}$  to  $\mathcal{N}$ ?

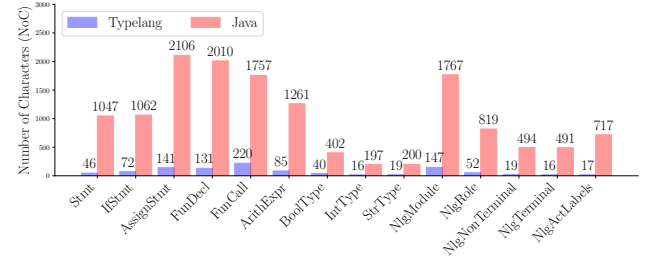
These questions assess our system’s modularity and reuse at the artifact level (*RQ<sub>1</sub>*), the degree of automation in LSP plugin generation (*RQ<sub>2</sub>*), and the feasibility of deriving language servers directly from *Typelang* variants (*RQ<sub>3</sub>*). Our case study proceeds as follows: we first implement a type system for several Neverlang artifacts using *Typelang*; then, we assemble these arti-

<sup>16</sup>VSCode is the most popular editor according to the Stack Overflow Developer Survey 2021 and it embraces the TextMate grammar implementing the native support for the syntax highlighting (see <https://github.com/microsoft/vscode-textmate>)





(a) A visualization of the LoC reduction achieved by implementing a type system using **Typelang** for some Neverlang artifact compared to a traditional Java-based approach. This figure illustrates how the use of **Typelang** leads to significant simplification, with a reduction of approximately 89.06%.



(b) A visualization of the NoC reduction achieved by implementing a type system using **Typelang** for some Neverlang artifact compared to a traditional Java-based approach. This figure illustrates how the use of **Typelang** leads to significant simplification, with a reduction of approximately 93.48%.

Figure 7: A comparison of the LoC and NoC needed to implement a type system using **Typelang** compared to a traditional Java-based approach.

Abstract Term	Neverlang Term
Language Artifact	Module
Feature Box	Annotated Java class
Collector	Annotation preprocessor
Type checking/inference semantics	Type checking/inference role

Table 3: Mapping the defined abstract terms to their counterparts in Neverlang.

facts into two complete language variants—SimpleLanguage,<sup>17</sup> a general-purpose language from the GraalVM project [142],<sup>18</sup> and Neverlang itself—and generate their language servers and LSP plugins for Visual Studio Code, NeoVim, and Vim. We quantify the effort in terms of lines of code (LoC) and number of characters (NoC), offering objective metrics for comparison against prior work [25]. The full source code and case study implementation are publicly available on Zenodo.<sup>19</sup> The following subsections describe the experimental setup, present results for each metric, and answer  $RQ_1$ ,  $RQ_2$ , and  $RQ_3$  based on our empirical findings.

### 6.1. Tool Support

To have a clear correspondence between the generic terms used previously and the specific Neverlang terms for those concepts, we provide a mapping in Table 3.

Our **Typelang** implementation leverages Neverlang and integrates with its parser [30] to use its polyglot capabilities [132]. Prior to the introduction of **Typelang** in Neverlang, developers had to write significant amounts of Java code to implement custom type checking and type inference rules, with limited reuse for existing type system specifications. For instance, Listing 7 demonstrates how the **Typelang** DSL can define type system rules for the `stmt.AssignStatement` module in Neverlang. In this example, the assignment is treated as a new declaration and will be statically and strongly typed.

**Neverlang Integration.** We extended the Neverlang compiler with an annotation preprocessor capable of handling the **Typelang** DSL. Neverlang modules can include semantic actions annotated with the `<typelang>` label; any code within

```

1 module stmt.AssignStatement {
2   imports { /* ... */ }
3
4   reference syntax {
5     AssignStatement ← Identifier "=" Expression ";" ;
6   }
7
8   role(execution) { /* assignment semantics */ }
9
10  role(check-infer) {
11    @<typelang> @{
12      try {
13        infer identifier $1.token with [$2.type] ⇒ $1.type
14        check $1.token : $1.type is invariant $2.type
15        use $1.token as $1.type
16      } on InferenceException {
17        define $2 $1
18      }
19    }.
20  }
21 }

```

Listing 7: An example of **Typelang** DSL to perform type checking and type inferencing for an assignment statement in the Neverlang language workbench.

these semantic actions will be written according to the syntax of a **Typelang** variant. When the Neverlang parser encounters the `<typelang>` annotation in a module, the Neverlang annotation preprocessor gathers all annotated Java classes to generate the **Typelang** variant needed to compile that specific Neverlang module. Neverlang delegates the parsing of the code of these semantic actions to the **Typelang** parser. If any features remain unrecognized after the collection phase, Neverlang will fail during parsing. Furthermore, Neverlang uses *dependency injection* (via the Google Guice Library [136]) to distribute the annotated Java class to internal structures. Through this injection, Neverlang ensures that all languages containing the `stmt.AssignStatement` module will also inherit its type checking and type inferencing capabilities.

**Assignment Statement Example.** Listing 7 shows the role `check-infer` (line 10) which uses its own **Typelang** variant. This variant is deduced from the feature boxes being used—specifically, `identifier` and `InferenceException` in this case. The `check-infer` role assumes the existence of a variant  $v_i$  from the **Typelang** family  $P$  capable of compiling the `stmt.AssignStatement` module; if such a variant exists, it will be used for type checking and type inferencing; otherwise the module cannot be compiled. The `check-infer` role uses the

<sup>17</sup><https://github.com/graalvm/simplelanguage>

<sup>18</sup><http://openjdk.java.net/projects/graal>

<sup>19</sup><https://doi.org/10.5281/zenodo.15276991>

keyword try to infer the type of the right-hand side expression (line 13) and checks that the types of the left-hand side and right-hand side are compatible (line 14). Line 15 declares the effective usage of the left-hand side identifier with the inferred type. If an *InferenceException* is raised during these phases, it indicates that the type associated with the identifier cannot be inferred. Consequently, line 17 is executed to define a new identifier whose type corresponds to the type of the right-hand side expression. This approach assumes that each time an *InferenceException* occurs, a *new variable declaration* rather than the *re-assignment* of an existing one is performed. While this may not always be true depending of the language (the exception could arise for several reasons, for instance, because the type of the right-hand side expression is not visible in the current scope, or the type of the right-hand side expression is not compatible with the type of the left-hand side identifier previously declared), but it can be readily covered. It is reasonable to assume that the `stmt.AssignStatement` module will be used in conjunction with another Neverlang module defining a declared *scope* and a *priority level*, as shown in Fig. 4.

**Typing Environment.** Each *typing environment* is associated to a scope defined by a Neverlang module. The *typing environment* admits the `stmt.AssignStatement` module will contain unordered pairs of the form (*ID*, *TableEntry*) for all identifiers declared within its scope, as shown by the blue boxes in Fig. 5. Given the variety of type inference strategies available—including, *Hindley-Milner* [63, 101], *constraint-based* [111], and *unification-based* [120] algorithms—Neverlang employs a strategy pattern [54] to select the algorithm for the type inference phase. Furthermore, additional and custom type inference strategies can be integrated later without modifying the existing ones, ensuring the type system’s extensibility.

**Reduction in LoC and NoC.** Implementing the check-infer role in Java required 64 LoC and 2, 106 NoC (excluding whitespace, newline, and tab characters). The same role in Typelang achieves a LoC reduction of approximately 89.06% (see Fig. 7a) and a NoC reduction of approximately 93.48% (see Fig. 7b), resulting in a total of 7 LoC and 141 NoC. The summary in Fig. 7 demonstrates that the Typelang DSL effectively reduces the amount of code needed to implement a type system for several artifacts in Neverlang. We calculated the overall LoC and NoC reduction percentages using the following formulas:

$$\text{LoC} = \frac{\text{LoC}_{\text{java}} - \text{LoC}_{\text{typelang}}}{\text{LoC}_{\text{java}}} \times 100$$

$$\text{NoC} = \frac{\text{NoC}_{\text{java}} - \text{NoC}_{\text{typelang}}}{\text{NoC}_{\text{java}}} \times 100$$

and the average per-artifact LoC and NoC percent reduction using the following formulas:

$$\overline{\text{LoC}} = \frac{\sum_{i=1}^n \text{LoC}_i}{n} \quad \overline{\text{NoC}} = \frac{\sum_{i=1}^n \text{NoC}_i}{n}$$

The results of our analysis are:

- overall and average LoC reduction percentage are approximately 82.90% and 82.32%, respectively;

```

1 bundle commons.expressions.Expressions {
2   slices
3   // Types
4   commons.types.IntegerType
5   commons.types.DoubleType
6
7   // Unary Expressions
8   commons.expressions.unary.UnaryNotSlice
9   commons.expressions.unary.UnaryPlusSlice
10  commons.expressions.unary.UnaryMinusSlice
11  commons.expressions.unary.UnaryComplementSlice
12  commons.expressions.unary.UnaryOperand
13  commons.expressions.unary.ParenthesizedExpression
14
15  // Additive Expressions
16  commons.expressions.add.AdditionSlice
17  commons.expressions.add.AdditionOperandSlice
18
19  // Multiplicative Expressions
20  commons.expressions.mul.MultiplicationSlice
21  commons.expressions.mul.MultiplicationOperandSlice
22 }
```

Listing 8: A Neverlang **bundle** to define the Expressions module.

- overall and average NoC reduction percentage are approximately 93.87% and 93.18%, respectively.

## 6.2. Degree of Type System Reuse

**Overview.** In recent decades, reuse has gained recognition as a key factor in software development, encompassing both software engineering [95, 57, 6, 21] and programming language design [96, 8, 67, 13]. This work aims to reduce the code needed to implement a programming language type system. This section discusses the extent of type system reuse within Neverlang and how Typelang facilitates it. The Typelang DSL empowers language developers to define type systems modularly, thereby enabling the reuse of these type systems across various languages.

To illustrate the degree of reuse of type systems in Neverlang, an example of a Neverlang **bundle** is needed. A **bundle**—shown in Fig. 8—is a Neverlang construct that contains a set of **slices** intended for inclusion within a **language** construct. In this example, the Expressions bundle defines the type system for expressions in Neverlang, specifically for *additive* and *multiplicative* expressions over integer and double types. A Neverlang **language** can import multiple **bundles**, enabling the reuse of type systems across different languages.

To demonstrate the extent of type system reuse in Neverlang, we implemented a set of *feature boxes* (see Fig. 8a) for:

- the most common primitive types, and
- the most common operators defined on these types.

Furthermore, we implemented a set of *semantic actions* for the Typelang roles in Neverlang for the Expressions (see Fig. 8b).

**Metrics for the Degree of Reuse.** This section aims to quantify the number of expression languages in which a type system can be reused. We define the following sets:

- *U*, a set of *N* primitive types,
- *O*, a set of *M* operators defined on these primitive types.

An expression language is defined by two subsets:  $u \subseteq U$  and  $o \subseteq O_U$ , where  $O_U = \{o \in O \mid \text{types}(o) \subseteq u\}$ .<sup>20</sup> Operators are dependent on types; an operator can only be included if it is defined on the types present in the expression language. Our strategy to count the number of expression languages involves counting the combinations of primitive types and the operators that can be defined on them. There are  $2^N$  possible combinations of primitive types. For each combination of primitive types, we can include only the operators defined on those types. Thus, for each  $u \subseteq U$  we can construct  $2^{|O_u|}$  combinations of operators, where  $|O_u|$  is the number of operators defined on the types in  $u$ . The total number of expression languages  $\mathcal{L}$  is given by:

$$\mathcal{L} = \sum_{u \subseteq U \mid u \neq \emptyset} (2^{|O_u|} - 1)$$

where  $-1$  is used to exclude the empty set of operators. That is, languages with no operators are not considered valid expression languages.

Consider a concrete example where  $U = \{int, double\}$  and  $O = \{+, *\}$  in which:

$$\begin{aligned} + : int \times int &\rightarrow int, & * : int \times int &\rightarrow int, \\ + : double \times double &\rightarrow double, & * : double \times double &\rightarrow double. \end{aligned}$$

The possible combinations of primitive types and operators are:

$$\begin{aligned} u = \{int, double\}, o = \{+, *\} & \quad u = \{int, double\}, o = \{*\} \\ u = \{int\}, o = \{+, *\} & \quad u = \{int\}, o = \{*\} \\ u = \{double\}, o = \{+, *\} & \quad u = \{double\}, o = \{*\} \\ u = \{int, double\}, o = \{+\} & \quad u = \{int, double\}, o = \emptyset \\ u = \{int\}, o = \{+\} & \quad u = \{int\}, o = \emptyset \\ u = \{double\}, o = \{+\} & \quad u = \{double\}, o = \emptyset \end{aligned}$$

The total number of expression languages ( $\mathcal{L}$ ) is calculated by:

$$\begin{aligned} \mathcal{L} &= (2^{|O_{\{int, double\}}|} - 1) + (2^{|O_{\{int\}}|} - 1) + (2^{|O_{\{double\}}|} - 1) \\ &= (2^{|\{+, *\}|} - 1) + (2^{|\{+\}|} - 1) + (2^{|\{*\}|} - 1) \\ &= (2^2 - 1) + (2^1 - 1) + (2^1 - 1) \\ &= 9 \end{aligned}$$

We introduce two metrics to evaluate the degree of type system reuse in Neverlang, drawing inspiration from the software reuse literature [83, 53, 10]: the normalized absolute reuse degree and the operator conditional reuse degree.

**Normalized absolute reuse degree (NAR).** It measures how many times a type system component is reused across the expression language implementation. It is defined as:

$$NAR(c) = \frac{|\{l \in L \mid c \in l\}|}{\mathcal{L}}$$

where  $c$  is a component (a type or an operator) of the type system, and  $L$  is the set of all expression languages.

**Operator conditional reuse degree (OCR).** It estimates the reuse of an operator within the expression languages that include

a specific primitive type. It is defined as:

$$OCR(o \mid t) = \frac{|\{l \in L \mid t \in l \wedge o \in l\}|}{|\{l \in L \mid t \in l\}|}$$

where  $t$  is a primitive type,  $o$  is an operator defined on it, and  $L$  is the set of all expression languages.

**Concrete Example.** Consider the set of expression language  $L = \{l_1, l_2, l_3\}$  where:

- $L_1 = \{int, +\}$ ,
- $L_2 = \{bool, int, +, *, ==\}$ ,
- $L_3 = \{double, +, *\}$ .

The NAR for each operator is calculated as:

$$\begin{aligned} - NAR(+) &= \frac{|\{l_1, l_2, l_3\}|}{3} = \frac{3}{3} = 1, \\ - NAR(*) &= \frac{|\{l_2, l_3\}|}{3} = \frac{2}{3} \approx 0.67, \text{ and} \\ - NAR(==) &= \frac{|\{l_2\}|}{3} = \frac{1}{3} \approx 0.33. \end{aligned}$$

On the other hand, the OCR for each operator is calculated as:

$$\begin{aligned} - OCR(+, int) &= \frac{|\{l_1, l_2\}|}{|\{l_1, l_2\}|} = \frac{2}{2} = 1, \\ - OCR(*, int) &= \frac{|\{l_2\}|}{|\{l_1, l_2\}|} = \frac{1}{2} = 0.5, \text{ and} \\ - OCR(==, int) &= \frac{|\{l_2\}|}{|\{l_1, l_2\}|} = \frac{1}{2} = 0.5. \end{aligned}$$

The LoC and NoC saved for the *feature boxes* and the *semantic actions* implementing the Typelang roles for each  $l \in L$  can be trivially calculated by summing respective values reported in Fig. 8a and Fig. 8b.

The NAR and OCR metrics provide a quantitative assessment of type system reuse. While NAR reflects the absolute reuse of a component across expression languages, OCR captures conditional reuse based on the presence of a given primitive type. Importantly, these metrics are not limited to primitive operators of the expression languages—they can be generalized to any programming language component defined in a modular way. That is, the NAR and OCR metrics offer a general method for evaluating type system reuse across modular language implementations. By using this approach, we also demonstrate that the number of combinations is reduced from  $\mathcal{T} \times \mathbf{1}$  to  $\mathcal{N} \times \mathbf{1}$ , where  $\mathcal{T} = \mathcal{L}$  represents the number of type systems and  $\mathcal{N} \ll \mathcal{T}$ .

### 6.3. Language Server Generation

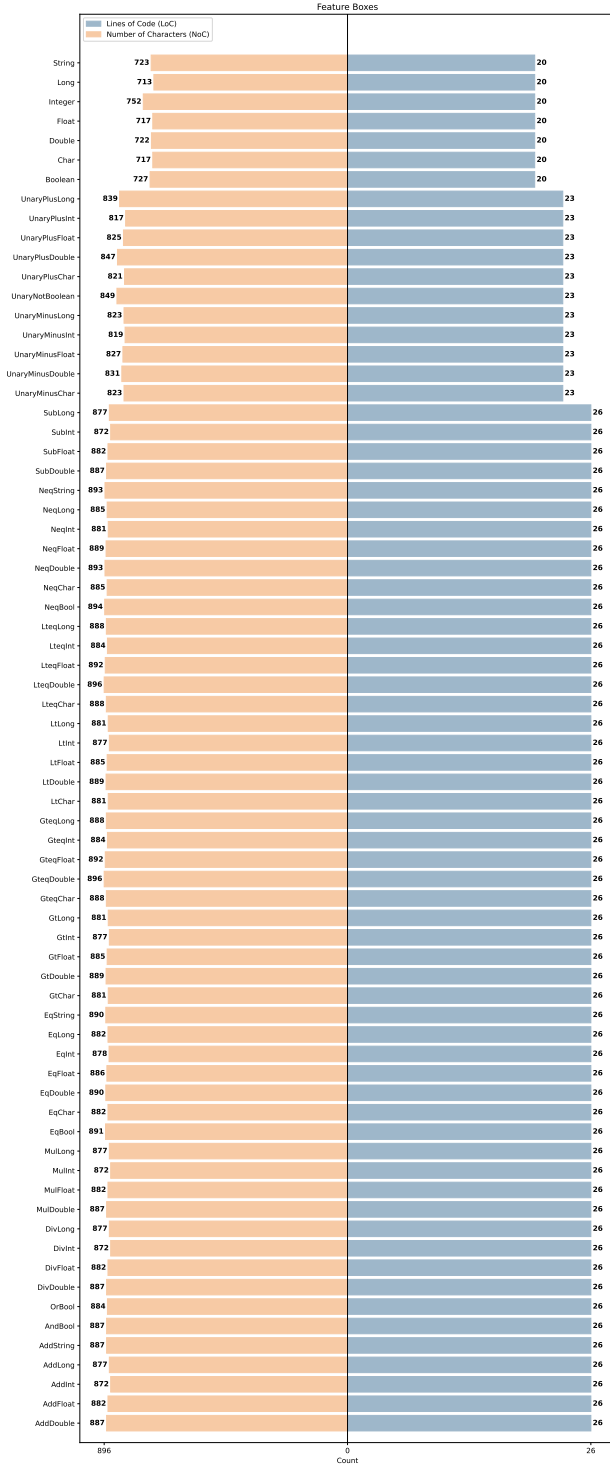
We developed a set of default feature boxes for Neverlang, implemented as annotated Java classes. These classes are designed to be used within the Typelang-related semantic actions to perform type checking and type inference. In particular, we provide:

- approximately 20 annotated Java classes for types;
- approximately 20 annotated Java classes for signatures;
- approximately 10 annotated Java classes for scopes;
- approximately 5 annotated Java classes for exceptions.

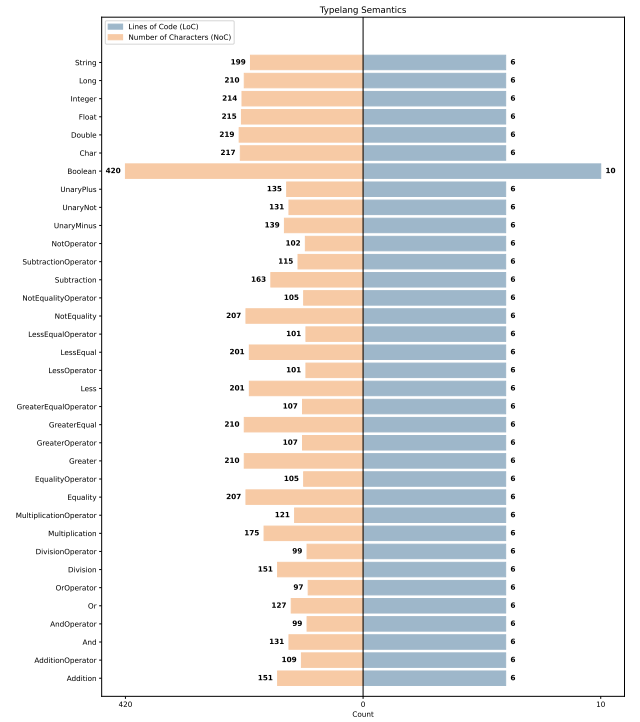
For each component described in Sect. 5.1 we provide a corresponding set of default implementations that conform to the respective interfaces. The orchestration and communication

<sup>20</sup>The types of an operator are the types on which the operator is defined.





(a) A visualization of the LoC and NoC needed to implement the *feature boxes* for the Java primitive types and the operators defined on them.



(b) A visualization of the LoC and NoC needed to implement the *semantic actions* for the Typelang roles in Neverlang. The Typelang roles are used to define the type system for the expressions in Neverlang.

Figure 8: A comparison of the LoC and NoC needed to implement a type system using Typelang compared to a traditional Java-based approach.

```

1 public class SimpleLangWorkspaceHandler
2     extends WorkspaceHandler {
3
4     public SimpleLangWorkspaceHandler(Workspace workspace) {
5         super(
6             workspace,
7             new DefaultIncrementalCompilationHelper());
8     }
9
10    public SourceSet getSourceSet(Path rootDir) {
11        return new DefaultSourceSet.Builder(".sl")
12            .buildFromRootDir(rootDir);
13    }
14
15    public Language language() {
16        return new SimpleLang(new SimpleLangModule());
17    }
18
19    public Stream<Role> lspRoles() {
20        return Stream.of(new LayeredRole(List.of(
21            new Role("check_infer", Role.Flags.MANUAL)));
22    }
23
24    public Class<?> extends AbstractCompilationHelper<?, ?>>
25        compilationHelper() {
26        return CompilationHelper.class;
27    }
28
29    public List<Priority> priorities() {
30        return List.of(new Sources(),
31            new File(),
32            new Function());
33    }
34
35 }

```

Listing 9: The *workspace handler* implementation for *SimpleLanguage*.

among these components are handled by the Neverlang runtime, relieving the language developer from the burden of implementing additional coordination logic. For the sake of completeness, the components for which we provide default implementations include:

- Compilation Unit,
- Compilation Unit Task,
- Compilation Unit Executor,
- Table Entry,
- Fenwick Tree,
- LSP Graph,
- Compilation Helper.

Language developers can choose to either adopt the provided default implementations or define custom ones from scratch to suit specific needs. This design allows language developers to obtain a fully functional language server out of the box, while still maintaining the flexibility to tailor it to their language’s requirements. The only exception is the *workspace handler*, for which Neverlang does not provide a default implementation. This is because the *workspace handler* handles pieces of information that are strictly related to the implemented language and cannot be predetermined. Instead, Neverlang provides an abstract class that developers can extend to implement the required behavior. Listing 9 shows an example *workspace handler* implementation for *SimpleLanguage*. An implementation for the *workspace handler* abstraction involves defining the methods:

- `getSourceSet` (line 10) returns the source set of the language based on the given file extension; that is, the collection of files that must be processed by the LSP;
- `language` (line 15) returns an instance of the language;

- `lspRoles` (line 19) returns a stream of roles associated to the LSP; *i.e.*, the names of all parse tree visitors needed to populate the LSP-related data structures;
- `compilationHelper` (line 24) returns the class of the concrete implementation of the compilation helper;
- `priorities` (line 29) returns a list of priorities for the compilation units, sorted according to the desired visit sequence.

Notice how the *workspace handler* connects all components and embodies the exogenous approach to dependencies management. For instance, by specifying priorities within the *workspace handler*, each priority definition remains unaware of the others, as well as of their relative order. Aside from the Neverlang code that defines the language—optionally leveraging the reusable default implementations—the *workspace handler* is the only component that must be implemented by the developer to obtain a fully functional language server for a given language, such as *SimpleLanguage* in this case.

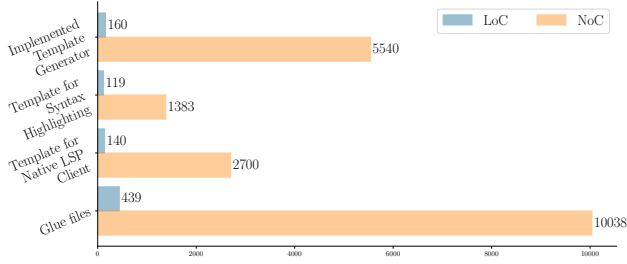
#### 6.4. LSP Client Generation

The LSP is completed by the *LSP client generator*. This generator is based on a system of *template generators* responsible for generating syntax highlighting rules and LSP configuration files for supported editors. To demonstrate the flexibility and simplicity of our approach, we implemented the aforementioned Gradle plugin, which currently supports Vim, NeoVim and VS-Code. Support for additional editors can easily be added in the future. Each plugin has been used to generate the corresponding LSP plugin for both *SimpleLanguage* and *Neverlang*. Fig. 9 shows the LoC and the NoC needed to implement the template generators for the three editors. The implementation is relatively straightforward: template strings contain placeholders that are filled with values specific to the implemented language. Each placeholder follows the format `${placeholder}`. Figs. 9a, 9b and 9c report the LoC and NoC for the VSCode, NeoVim, and Vim plugins respectively, grouped by the following categories:

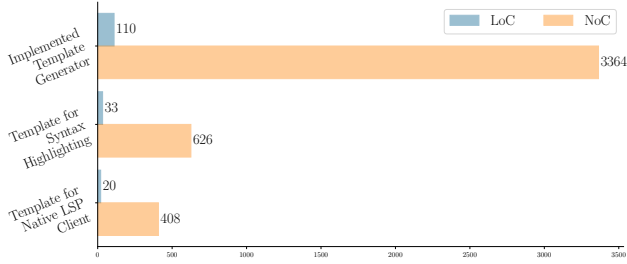
- *implemented template generator*, the Java code, integrated with the Gradle plugin, that implements the template generator;
- *template for syntax highlighting*, the annotated template defining syntax highlighting;
- *template for LSP plugin*, the annotated template for the LSP plugin;
- *glue files*, additional files required only for full support in VSCode.

Each template needs to be annotated only once per editor. After that, the Gradle plugin uses language-specific input to populate the placeholders and generate a concrete plugin for any supported language.

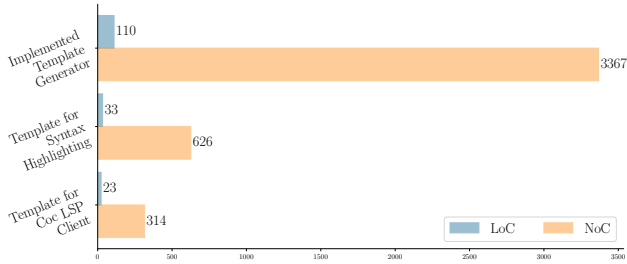
Note that in our analysis, the NoC excludes whitespace, new-line and tab characters. According to Fig. 9a, the *implemented template generator* for VSCode has been implemented in 160 LoC and 5,540 NoC. For Vim (Fig. 9c) and NeoVim (Fig. 9b), the template generators have been implemented in 110 LoC and 3,364 NoC. Vim and NeoVim share the same file format for syntax highlighting, we could reuse the same code for both (see



(a) LoCs and NoCs required to implement the template generators for VSCode. The VSCode generator leverages the existence of annotated templates for syntax highlighting and LSP plugin. Additional glue files specific to VSCode are needed to complete the support.



(b) LoCs and NoCs required to implement the template generators for NeoVim. The NeoVim generator leverages the existence of annotated templates for syntax highlighting and LSP plugin.



(c) LoCs and NoCs required to implement the template generators for Vim. The Vim generator leverages the existence of annotated templates for syntax highlighting and LSP plugin.

Figure 9: Comparative analysis of LoCs and NoCs required to implement the template generators for VSCode, NeoVim and Vim. The figure highlights the differences in implementation complexity, demonstrating the flexibility and adaptability of the Gradle plugin in supporting multi editor ecosystems.

Fig. 9b and Fig. 9c). Moreover, NeoVim has built-in support for the LSP protocol, whereas Vim needs an external plugin. We chose to adopt CoC (*Conquer of Completion*), one of the most widely used LSP clients for Vim. However, similar template generators could be created for other Vim plugins with minimal effort. In our implementation, using CoC required slightly less code (20 LoC) than supporting the native NeoVim client (23 LoC). According to Bänder and Kuchen [25], the development effort for VSCode support is significant (around 725 minutes), as is the effort for implementing the language server (around 350 minutes). These figures are consistent with our findings: the amount of code required for the VSCode client is noticeably higher compared to the other editors. Specifically, the *template for syntax highlighting* consists of 119 LoC and 1,383 NoC, while the *template for LSP plugin* comprises 140 LoC and 2,700 NoC.

## 6.5. Discussion

This section presents the results of the case study and addresses the research questions introduced in Sect. 1. We then discuss the limitations of our approach and its applicability to other language workbenches. Finally, we explore the implications of our approach for the future of the LSP.

### 6.5.1. Research Questions

In this study, we presented a demonstration case study to showcase the effectiveness of our approach. We addressed the three research questions introduced in Sect. 1, providing a detailed answer to each.

*RQ<sub>1</sub> To what degree is it possible to streamline by associating variants to language artifacts?*

We demonstrated that by associating Typelang variants with language artifacts, it is possible to define type checking and type inference semantics in a modular way. The *variant-oriented programming* paradigm, along with the *cross-artifact coordination* layer, played a key role in enabling different variants to coexist and span across multiple language artifacts. This results in a flexible and modular type system. The benefits of this modularization are reflected in our results: the amount of code required to implement a type system was reduced by approximately 82.90% in terms of LoC and 93.87% in terms of NoC. On a per-artifact basis, the average reduction was approximately 82.32% for LoC and 93.18% for NoC. Moreover, the developed artifacts are reusable across different languages, which further reduces the overall development effort.

*RQ<sub>2</sub> To what degree is it possible to automate the generation of LSP clients, lowering  $\mathcal{E}$  to 1?*

LSP clients for different editors can be generated using the Gradle plugin introduced in Sect. 5.4. This plugin is responsible for generating both syntax highlighting and LSP configuration files for the supported editors. The amount of code needed to implement the template generators varies across editors: the VSCode generator needs 160 LoC and 5,540 NoC, while the NeoVim and Vim generators need 110 LoC and 3,364 NoC. The higher code requirements for the VSCode generator stem from the need for additional glue files to achieve full language support. In contrast, NeoVim provides native support for the LSP protocol, and Vim generators need less code because NeoVim offers a native support for the LSP protocol, meanwhile Vim relies on an external plugin. Since NeoVim uses the same syntax highlighting format as Vim, the same code can be reused for both. Each template needs to be annotated only once per editor, after which the plugin can be reused to generate LSP clients for all supported editors and for any language. As a result, the effort required to support multiple editors is effectively reduced from  $\mathcal{E}$  to 1, thanks to the automation provided by the plugin.

*RQ<sub>3</sub> Can the language server be automatically generated starting from the Typelang variants lowering  $\mathcal{L}$  to  $\mathcal{N}$ ?*



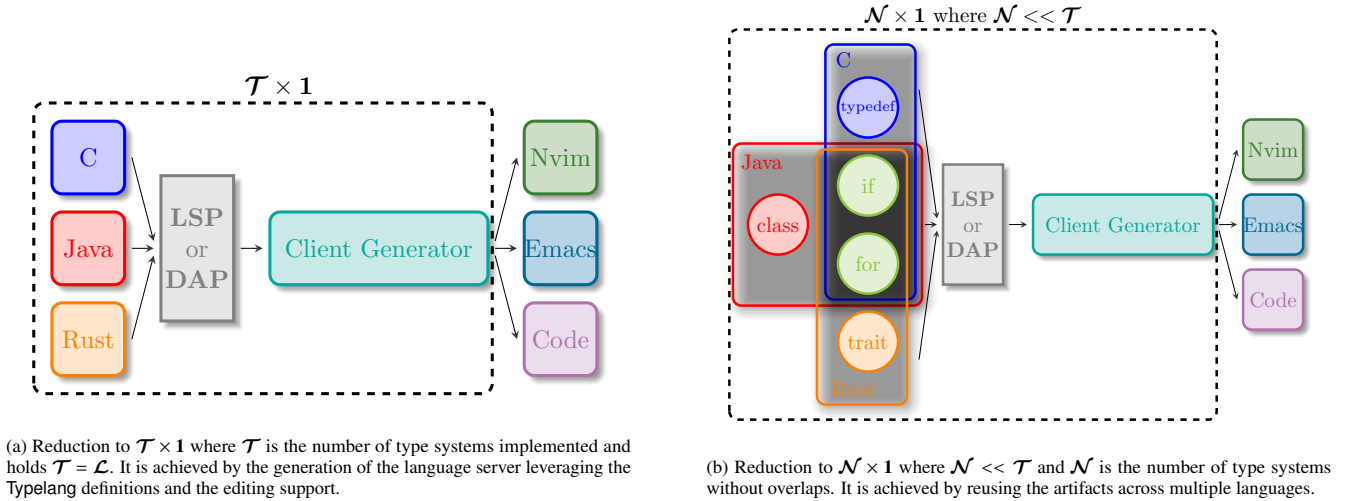


Figure 10: A graphical representation of the  $\mathcal{T} \times 1$  and  $\mathcal{N} \times 1$  reduction using the client generator.

In Sect. 5.1 and Sect. 5.3, we highlighted the possibility of generating the language server implementation by leveraging the data structures provided by the language workbench. We also demonstrated that the properties of *variant-oriented programming* and *cross-artifact modularization* layer have been crucial in ensuring that these data structures can be accessed across different variants. By implementing the language servers according to the proposed methodology, we effectively reduced the number of language server implementations from  $\mathcal{L}$  to  $\mathcal{N}$  where  $\mathcal{N} \ll \mathcal{L}$ . The first reduction—from  $\mathcal{L} \times 1$  to  $\mathcal{T} \times 1$ , as shown in Fig. 10a—is achieved by generating the language server from Typelang definitions. Even though  $\mathcal{L} = \mathcal{T}$ , this results in a significant reduction in development time and effort, as defining the type system in Typelang is substantially simpler than implementing a full language server. This is supported by our case study and the answer to  $\mathcal{RQ}_1$ . To the best of our knowledge, the best reduction in the literature so far is  $\mathcal{T} \times \mathcal{E}$ , as achieved by Xtext. However, Xtext enforces a monolithic implementation of languages and does not support automated generation of LSP client plugins, making it impossible to reduce  $\mathcal{E}$  and, consequently, offering no gains on the client side. Client implementation is often overlooked in LSP development, yet it can represent a significant portion of the effort, as shown by Bänder and Kuchen [25]. Our proposal not only improves upon the current state of the art by reducing the cost to  $\mathcal{T} \times 1$ , but it also achieves a further reduction to  $\mathcal{N} \times 1$ , where  $\mathcal{N} \ll \mathcal{T}$  represents the number of distinct, non-overlapping type systems. This is made possible through artifact reuse across multiple languages, as illustrated in Fig. 10b.

### 6.5.2. Limitations

In this section, we discuss the limitations of our approach to generating LSP clients and language servers. Specifically, we address the limitations concerning: i) the proposed data structures (i.e., the *Fenwick Tree* and the *LSP Graph*), and ii) the Gradle plugin.

The *Fenwick tree* and the *LSP graph* are data structures introduced in the context of LSP. While the former is a well-

established data structure, it is not commonly used in LSP scenarios and could be replaced with alternatives such as the *segment tree* [12]. The segment tree is less efficient in terms of memory usage— $O(2n)$  instead of  $O(n)$ —and has slightly higher time complexity— $\log n + k$  instead of  $\log n$ , where  $n$  is the number of elements and  $k$  the number of retrieved segments. However, unlike the Fenwick tree, which supports only prefix sums or point updates, the segment tree can support any associative function (e.g., sum, min, max, gcd) and enables efficient range updates via lazy propagation.

The *LSP graph* is a novel data structure introduced specifically for the LSP domain. To the best of our knowledge, no alternative data structure currently exists that serves the same purpose. However, our implementation does not yet support *code completion*—a feature that can be trivially derived from the LSP Graph. For example, the neighbors of a node in the graph can serve as valid completion suggestions. We chose not to implement this feature in our case study, as it was not strictly necessary for demonstration purposes; nonetheless, it is a clear direction for future work.

Typelang is a domain-specific language designed to define type systems within the LSP context. It is not a general-purpose language and is not intended for use outside the domain of type system definition. However, as a family of DSLs, a full implementation of Typelang could be used to define type systems for a wide range of programming languages. By leveraging the *variant-oriented programming* paradigm and the *cross-artifact coordination* layer, developers can reuse type system components across multiple languages.

The Gradle plugin is a specialized tool designed for LSP-related tasks. Although it is not intended as a general-purpose Gradle plugin, it can be used to generate LSP clients for various programming languages, as demonstrated in our case study. At present, it supports a limited number of editors, but it is easily extensible to support additional ones.

### 6.5.3. Applicability

The applicability of our approach extends beyond the Neverlang language workbench. It is suitable for any language workbench that supports the modularization of linguistic components. As shown in Table 1, other popular language workbenches—such as Melange, MPS, and Spoofox—also support this feature. Therefore, our approach can be applied to these language workbenches as well.

Both the *variant-oriented programming* paradigm and the *cross-artifact coordination* layer are generic mechanisms, applicable to any programming language. From one perspective, Typelang is a DSL designed to demonstrate the effectiveness of these paradigms; from another, it is a DSL for implementing type systems. Any DSL that supports the modularization of linguistic components could replace Typelang without compromising the benefits provided by our approach.

The same holds true for the Gradle plugin. While originally developed for Neverlang, its applicability is not restricted to language workbenches with modular linguistic components. Instead, it can be used with any language workbench that supports generating a compiler or interpreter targeting the JVM. Extending the plugin to support non-JVM-based languages is part of our planned future work. In conclusion, our approach is not confined to Neverlang: it generalizes to any language workbench with support for linguistic modularization and JVM-based compilation or interpretation.

### 6.5.4. Implications

The progressive reduction from  $\mathcal{L} \times \mathcal{E}$  to  $\mathcal{T} \times \mathbf{1}$ , and ultimately to  $\mathcal{N} \times \mathbf{1}$ , carries significant implications for the future of the LSP. By minimizing the number of required language servers and clients, our approach simplifies the development process and reduces the inherent complexity of LSP implementations. This simplification translates into faster development cycles, lower maintenance overhead, and improved runtime performance for both servers and clients.

Moreover, this reduction facilitates greater modularity and reusability of components across tools and languages. Developers can focus on implementing language-specific logic once, rather than repeatedly adapting it to fit a matrix of editors and environments. This not only lowers the barrier to entry for supporting new languages but also enhances the consistency and reliability of the user experience across tools.

From an ecosystem perspective, our model supports a more sustainable and scalable architecture. As the number of programming languages and development tools continues to grow, a leaner interaction model helps prevent the combinatorial explosion in LSP implementations. It also paves the way for richer interoperability and tool composition, potentially enabling new workflows and integrations that were previously too costly or complex to implement.

We believe that this streamlined architecture has the potential to make LSP more approachable, practical, and powerful for developers. In the long term, it may pave the way for a more unified and efficient tooling landscape across programming languages and environments.

## 7. Related Work

**Programming Paradigms.** Programming paradigms have long been an established topic of research in the software engineering community [118, 29, 68, 89, 66]. Many paradigms [113, 76] are specifically designed to address core software development concerns such as modularity, reusability, and maintainability. In *feature-oriented programming* [4, 41, 114], a feature module is a unit of composition that encapsulates specific functionality. It is treated as a first-class entity and can be composed with other feature modules to build complete software systems. Feature-oriented programming is typically employed in the development of software product lines and for the incremental development of programs. Similarly, *delta-oriented programming* [127, 42, 80] focuses on the dynamic and incremental application of changes (called deltas) to a core module. Both paradigms share conceptual similarities with our approach in that they aim to modularize software systems into units that encapsulate distinct functionality. However, our approach required the ability to explicitly define and compose variants within a complex software system. This need is not fully addressed by either feature-oriented or delta-oriented paradigms, as they are primarily concerned with the incremental evolution of programs or the modification of a central core module. In contrast, our variants are not necessarily incremental nor must they relate to a single core; moreover, the overall system may or may not form a product line.

A related paradigm is *aspect-oriented programming* [75, 90], which introduces aspect modules to encapsulate crosscutting concerns—those that affect multiple parts of a system. Aspects define pointcuts and advices, where pointcuts identify join points in the execution of a program and advices specify the behavior to execute at those points. While the similarities to our approach are subtle, our notion of shared contexts can be considered analogous to join points: they define the interaction boundaries where variants can communicate and coordinate.

**Variability in Language Variants.** de Lara et al. [92] employ graph transformation techniques to model and compose language variants. While their approach excels at visualizing and managing interdependencies among language components, it does not directly address the automation of type system integration or the generation of LSPs. In contrast, Typelang leverages a variant-oriented programming paradigm and a cross-artifact coordination layer to modularize type system definitions. These definitions are then automatically integrated into the language server generation process, thereby significantly reducing manual effort across different editors. de Lara and Guerra [91] adopt multi-level modeling to capture variability within language families, facilitating the systematic reuse of language artifacts. While their framework provides robust support for language evolution and family engineering, it is primarily focused on high-level models and lacks dedicated mechanisms for modularizing type system components or automating LSP plugin generation. Typelang addresses this gap by providing a family of DSLs that not only encapsulate type definitions and checking rules in a modular fashion but also directly support

the generation of editor integrations across multiple language artifacts.

*FunKons* [104], developed within the K framework, proposes a component-based approach to achieve modularity in semantics. Its main strength lies in composing semantic rules from independent components to ensure reusability at the semantic level. While *Typelang* shares the goal of modularity, it extends the focus to type systems, offering integrated support for type inference, type checking, and error reporting within a modular architecture that also facilitates LSP generation. Mosses [102] promote a modular formulation of operational semantics using structured rules. Though this work significantly advances the treatment of semantic variability, its scope remains confined to operational semantics. In contrast, *Typelang* captures semantic variability within its DSLs dedicated to type system definitions, addressing not only the correctness of language behavior but also the practical integration of type systems into development environments.

**Multi Product Lines.** In the context of software product lines [37, 112, 84], the *multi-product line* methodology [123, 124, 125] refers to a special case in which multiple product lines are integrated into a unified product line. Our approach aims to operationalize this concept by offering a programming paradigm that enables the specification and composition of multiple variants within a complex software system, which may itself be a product line. Several methodological approaches have been proposed to support the development of multi-product lines. Notably, in 2011, El-Sharkawy et al. [48] introduced a methodology to support the heterogeneous composition of multi-product lines. Similarly, Hartmann and Trew [60] proposed the *Context Variability Model*, a technique designed to constrain feature models, thereby enabling multi-product line support and facilitating staged configuration within software supply chains.

**Language Workbenches and IDEs.** The development of DSLs has become a prominent area of research, attracting considerable attention from the software engineering community [98, 79]. Several language workbenches—such as Spoofax [139], MPS [138], MontiCore [82], and Melange [45]—have been proposed to facilitate the creation of DSLs. These platforms typically support the generation of IDE features; however, such support is often generic and fails to leverage the specific characteristics of individual DSLs. Most rely on static templates that overlook the modularity of DSL features, making it challenging to specify and reuse IDE services in a composable fashion, as enabled by our approach. While MontiCore [26, 27] and Melange [97] offer support for LPLs, their IDE support is generally limited to basic features like syntax highlighting and code completion. EMFText [62] also deserves mention as an EMF-based tool [129]—similar to Xtext—that supports modular language development and facilitates IDE generation. Although EMFText does not explicitly address IDE variability, it shares several foundational concepts with our LPL-driven IDE approach, such as the use of attribute grammars to propagate information between languages and their IDEs, and a dedicated DSL for IDE description. Among available workbenches, Xtext [24] remains, to the best of our knowledge, the

only tool that supports automatic generation of language servers conforming to the LSP. However, Xtext is monolithic and does not support modularity in the development of language syntax, semantics, type systems, or language server implementations, as summarized in Table 1. More recently, Mosses [103] has proposed a formal, component-based methodology for building language workbenches that emphasizes correctness and reuse. While their framework systematically supports language development, it does not address the modularization of type systems or the generation of editor support. In contrast, *Typelang* is designed to operate within a modular language workbench, offering reusable type system fragments and enabling the automated generation of LSP clients for multiple editors. Langium [70], a modern monolithic language engineering tool, also supports LSP generation through a grammar-centric design. Although it provides robust LSP integration, it does not natively support modular type system definitions or the reuse of language components. *Typelang*, in contrast, introduces a family of DSLs designed specifically to modularize type systems and streamline the generation of LSP plugins for various editors, thereby enhancing both flexibility and reusability in language engineering.

**Languages for Type Systems.** Bettini et al. [18, 20, 19] introduced XSemantics, a DSL for specifying type systems and operational semantics in XText. Their work demonstrates how a language’s type system can enhance language server generation. While their focus is on improving language server generation through the type system, our approach extends this idea by modularizing both the type system and the language server implementation, providing greater flexibility and reusability.

Béguet and Amiard [11] employ satisfiability modulo theories (SMT) to express and verify type system properties. Their approach uses logical DSLs to ensure type rules adhere to soundness criteria, with the SMT solver validating these constraints. In contrast, *Typelang* is designed to promote modularity, composability, and reusability within a language workbench. Instead of focusing on automated reasoning about type correctness, *Typelang* integrates type system definitions into the development pipeline, automatically generating language server support and streamlining editor integration. This approach is particularly useful for environments where rapid reassembly and deployment of language artifacts across multiple editors are required.

In “Type Systems as Macros,” Chang et al. [36] treat type system definitions as syntactic extensions, allowing for automatic expansion of type constructs during compilation. While macros offer abstraction and reuse, their expansion is often closely tied to compiler infrastructure, limiting portability across different language workbenches or editors. *Typelang*, by contrast, is built around modular type system implementation through DSLs. This approach not only encapsulates type inference and checking but also integrates them into variant-oriented programming, enabling seamless composition of language artifacts and automatic generation of language server plugins.

Pacak et al. [109] focus on performance optimizations for type checking, particularly through incremental computation. Their method reduces recomputation by updating only the affected portions of the type checker. While this is essential for



scaling type checking in large codebases and enhancing interactive development, it does not align directly with the goals of Typelang. Typelang’s primary focus is on the reuse of type system components and the automation of language server generation. The variant-oriented design in Typelang emphasizes modularity and cross-artifact coordination, which could potentially incorporate incremental techniques as an optimization, though the core strength of Typelang lies in reusing type system fragments across diverse language variants.

Language workbenches like Spoofox and MPS provide end-to-end tooling for language development, including syntax definition, semantic analysis, and editor integration. However, they often suffer from monolithic type system implementations that are tightly coupled to the host environment. For example, MPS supports projectional editing and reusability at the syntax level but lacks dedicated mechanisms for modularizing type system definitions. Similarly, Spoofox, while powerful in transformations, does not offer explicit constructs for composing and reusing type checking rules. In contrast, Typelang addresses these limitations by introducing a family of DSLs that modularize type definitions, type inference, and type checking. Its variant-oriented programming model separates concerns and promotes the reuse of common type system components. Moreover, its integration with automated LSP plugin generation significantly reduces the development overhead of supporting multiple editors.

## 8. Conclusion

In this paper, we introduced Typelang, a family of DSLs for modular, composable, and reusable type system development. Leveraging *variant-oriented programming* and a *cross-artifact modularization* layer, Typelang enables the reuse of type system variants across artifacts and supports automated generation of language servers in modular workbenches. To reduce the LSP plugin implementation effort from  $\mathcal{E}$  to  $\mathbf{1}$ , we developed a Gradle plugin that automates plugin generation, reducing the effort further to  $\mathcal{T} \times \mathbf{1}$  and then to  $\mathcal{N} \times \mathbf{1}$ , where  $\mathcal{N} \ll \mathcal{T}$  through artifact reuse. Our approach streamlines the creation of type systems and LSP plugins for language families, improving efficiency in modular language workbenches. Empirical results show that our approach can reduce the effort required to implement type systems by 93.48% in terms of NoC and LSP plugins by 100%.

## Acknowledgments

This work was partly supported by the MUR project “T-LADIES” (PRIN 2020TL3X8X).

## References

[1] Aho, A.V., Sethi, R., Ullman, J.D., 1986. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, Reading, Massachusetts.

[2] Ammann, U., 1978. Error Recovery in Recursive Descent Parsers and Run-Time Storage Organization. Technical Report D-INFK. ETH Zürich, Department of Computer Science. Zürich, Switzerland.

[3] Apel, S., Leich, T., Saake, G., 2008. Aspectual Feature Modules. *IEEE Transactions on Software Engineering* 34, 162–180.

[4] Apel, S., von Thein, A., Wendler, P., Größlinger, A., Beyer, F., 2013. Strategies for Product-Line Verification: Case Studies and Experiments, in: Chang, B.H., Pohl, K. (Eds.), *Proceedings of the 35th International Conference on Software Engineering (ICSE’13)*, IEEE, San Francisco, CA, USA. pp. 482–491.

[5] Barros, D., Peldszus, S., Assunção, W.K.G., Berger, T., 2022. Editing Support for Software Languages: Implementation Practices in Language Server Protocols, in: Wimmer, M. (Ed.), *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems (MoDELS’22)*, ACM, Montréal, Canada. pp. 232–243.

[6] Bassett, P., 1996. *Framing Software Reuse: Lessons from the Real World*. Prentice Hall.

[7] Basten, B., van den Bos, J., Hills, M., Klint, P., Lankamp, A., Lissner, B., van der Ploeg, A., van der Storm, T., Vinju, J., 2015. Modular Language Implementation in Rascal—Experience Report. *Science of Computer Programming* 114, 7–19.

[8] Batory, D., Lofaso, B., Smaragdakis, Y., 1998. JTS: Tools for Implementing Domain-Specific Languages, in: *Proceedings of the 5th International Conference on Software Reuse (ICSR’98)*, IEEE Computer Society, Victoria, BC, Canada. pp. 143–153.

[9] Batory, D., Sarvela, J.N., Rauschmayer, A., 2004. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering* 30, 355–371.

[10] Bay, T.G., Pauls, K., 2004. Reuse Frequency as Metric for Component Assessment. Technical Report 464. ETH, Department of Computer Science. Zürich, Switzerland.

[11] Béguet, R., Amiard, R., 2023. Application of SMT in a Meta-Compiler: A Logic DSL for Specifying Type Systems, in: Graham-Lengrand, S., Preiner, M. (Eds.), *Proceedings of the 21st International Workshop on Satisfiability Modulo Theories (SMT’23)*, CEUR, Haifa, Israel. pp. 46–61.

[12] de Berg, M., Cheong, O., van Kreveld, M., Overmars, M., 2008. *Computational Geometry: Algorithms and Applications*. 3rd ed., Springer.

[13] Bertolotti, F., Cazzola, W., Favalli, L., 2023a. On the Granularity of Linguistic Reuse. *Journal of Systems and Software* 202. doi:10.1016/j.jss.2023.111704.

- [14] Bertolotti, F., Cazzola, W., Favalli, L., 2023b. SPJL92: Software Product Lines Extraction Driven by Language Server Protocol. *Journal of Systems and Software* 205. doi:10.1016/j.jss.2023.111809.
- [15] Bettini, L., 2011. A DSL for Writing Type Systems for Xtext Languages, in: Probst, C.W., Wimmer, C. (Eds.), *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java (PPPJ'11)*, ACM, Kongens Lyngby, Denmark. pp. 31–40.
- [16] Bettini, L., 2013a. Implementing Domain-Specific Languages with Xtext and Xtend. PACKT Publishing Ltd.
- [17] Bettini, L., 2013b. Implementing Java-like Languages in Xtext with Xsemantics, in: *Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC'13)*, ACM, Coimbra, Portugal. pp. 1559–1564.
- [18] Bettini, L., 2016. Implementing Type Systems for the IDE with Xsemantics. *Journal of Logical and Algebraic Methods in Programming* 85, 655–680.
- [19] Bettini, L., 2019. Type Errors for the IDE with Xtext and Xsemantics. *Journal Open Computer Science* 9, 52–79.
- [20] Bettini, L., von Pilgrim, J., Reiser, M.O., 2016. Implementing a Typed Javascript and Its IDE: A CASE-Study with Xsemantics. *Journal on Advances in Software* 9, 283–303.
- [21] Bezerra, C.I.M., Andrade, R.M.C., Monteiro, J.M., 2015. Measures for Quality Evaluation of Feature Models, in: *Proceedings of the 9th International Conference on Software and Software Reuse (ICSR'15)*, Springer, Miami, FL, USA. pp. 282–297.
- [22] Booch, G., Maksimchuk, R., Engle, M., Young, B., Conallen, J., Houston, K., 2007. *Object-Oriented Analysis and Design with Applications*. Third ed., Addison-Wesley.
- [23] Briand, L.C., Daly, J.W., Wüst, J., 1999. A Unified Framework for Coupling Measurement in Object-Oriented Systems. *IEEE Transactions on Software Engineering* 25, 91–121.
- [24] Bündler, H., 2019. Decoupling Language and Editor: The Impact of the Language Server Protocol on Textual Domain-Specific Languages, in: Hammoudi, S., Ferreira Pires, L., Selić, B. (Eds.), *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development (MODELWARD'19)*, SciTe Press, Prague, Czech Republic. pp. 129–140.
- [25] Bündler, H., Kuchen, H., 2019. Towards Multi-editor Support for Domain-Specific Languages Utilizing the Language Server Protocol, in: Hammoudi, S., Pires, L.F., Selić, B. (Eds.), *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development (MODELWARD'19)*, Springer, Prague, Czech Republic. pp. 225–245.
- [26] Butting, A., Eikermann, R., Kautz, O., Rumpe, B., Wortmann, A., 2018a. Controlled and Extensible Variability of Concrete and Abstract Syntax with Independent Language Features, in: *Proceedings of the 12th International Workshop on Variability Modelling of Software Intensive Systems (VAMOS'18)*, ACM, Madrid, Spain. pp. 75–82.
- [27] Butting, A., Eikermann, R., Kautz, O., Rumpe, B., Wortmann, A., 2018b. Modeling Language Variability with Reusable Language Components, in: Berger, T., Borba, P. (Eds.), *Proceedings of the 22nd International Systems and Software Product Line Conference (SPLC'18)*, ACM, Gothenburg, Sweden. pp. 65–75.
- [28] Cardelli, L., 1988. Structural Subtyping and the Notion of Power Type, in: Ferrante, J., Peter, M. (Eds.), *Proceedings of the 15th ACM Symposium on Principles of Programming Languages (POPL'88)*, ACM, San Diego, CA, USA. pp. 70–79.
- [29] Cardozo, N., Günther, S., D'Hondt, T., Mens, K., 2011. Feature-Oriented Programming and Context-Oriented Programming: Comparing Paradigm Characteristics by Example Implementations, in: Hartmann, H., Breivold, H.P. (Eds.), *Proceedings of the 6th International Conference on Software Engineering Advances (ICSEA'11)*, IARIA, Barcelona, Spain. pp. 130–135.
- [30] Cazzola, W., 2012. Domain-Specific Languages in Few Steps: The Neverlang Approach, in: Gschwind, T., De Paoli, F., Gruhn, V., Book, M. (Eds.), *Proceedings of the 11th International Conference on Software Composition (SC'12)*, Springer, Prague, Czech Republic. pp. 162–177.
- [31] Cazzola, W., Favalli, L., 2022. Towards a Recipe for Language Decomposition: Quality Assessment of Language Product Lines. *Empirical Software Engineering* 27. doi:10.1007/s10664-021-10074-6.
- [32] Cazzola, W., Favalli, L., 2024. Software Modernization Powered by Dynamic Language Product Lines. *Journal of Systems and Software* 218. doi:10.1016/j.jss.2024.112188.
- [33] Cazzola, W., Olivares, D.M., 2016. Gradually Learning Programming Supported by a Growable Programming Language. *IEEE Transactions on Emerging Topics in Computing* 4, 404–415. doi:10.1109/TETC.2015.2446192. special Issue on Emerging Trends in Education.
- [34] Cazzola, W., Poletti, D., 2010. DSL Evolution through Composition, in: *Proceedings of the 7th ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'10)*, ACM, Maribor, Slovenia.
- [35] Cazzola, W., Vacchi, E., 2013. Neverlang 2: Componentised Language Development for the JVM, in: Binder, W., Bodden, E., Löwe, W. (Eds.), *Proceedings of the 12th International Conference on Software Composition (SC'13)*, Springer, Budapest, Hungary. pp. 17–32.

- [36] Chang, S., Knauth, A., Greenman, B., 2017. Type Systems as Macros, in: Gordon, A.D. (Ed.), Proceedings of the 44th Symposium on Principles of Programming Languages (PoPL'17), ACM, Paris, France. pp. 694–705.
- [37] Clements, P., Northrop, L., 2001. Software Product Lines: Practices and Patterns. Addison-Wesley.
- [38] Cook, W.R., Hill, W., Canning, P.S., 1990. Inheritance Is Not Subtyping, in: Allen, F.E. (Ed.), Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'90), ACM, San Francisco, CA, USA. pp. 125–135.
- [39] Cooper, K.D., Torczon, L., 2022. Engineering a Compiler. Morgan Kaufmann.
- [40] Crane, M.L., Dingel, J., 2005. UML vs. Classical vs. Rhapsody Statecharts: Not All Models Are Created Equal, in: Briand, L., Williams, C. (Eds.), Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS'05), Springer, Montego Bay, Jamaica. pp. 97–112.
- [41] Czarnecki, K., Helsen, S., Eisenecker, U., 2004. Staged Configuration Using Feature Models, in: Weiss, D., van Ommering, R. (Eds.), Proceedings of the 3rd International Conference on Software Product-Line (SPLC'04), Springer, Boston, MA, USA. pp. 266–283.
- [42] Damiani, F., Schaefer, I., Winkelmann, T., 2014. Delta-Oriented Multi Software Product Lines, in: Heymans, P., Rubin, J. (Eds.), Proceedings of 18th International Software Product Line Conference (SPLC'14), ACM, Florence, Italy. pp. 232–236.
- [43] Damini, F., Lienhardt, M., Paolini, L., 2019. A Formal Model for Multi Software Product Lines. Science of Computer Programming 172, 203–231.
- [44] Deelstra, S., Sinnema, M., Bosch, J., 2005. Product Derivation in Software Product Families: A Case Study. Journal of Systems and Software 74, 173–194.
- [45] Degueule, T., Combemale, B., Blouin, A., Barais, O., Jézéquel, J.M., 2015. Melange: a Meta-Language for Modular and Reusable Development of DSLs, in: Di Ruscio, D., Völter, M. (Eds.), Proceedings of the 8th International Conference on Software Language Engineering (SLE'15), ACM, Pittsburgh, PA, USA. pp. 25–36.
- [46] Del Sole, A., 2023. Visual Studio Distilled. Third ed., Apress.
- [47] Ekman, T., Hedin, G., 2007. The JastAdd System — Modular Extensible Compiler Construction. Science of Computer Programming 69, 14–26.
- [48] El-Sharkawy, S., Kröher, C., Schmid, K., 2011. Supporting Heterogeneous Compositional Multi Software Product Lines, in: Santana de Almeida, E., Kishi, T. (Eds.), Proceedings of the 15th International Software Product Line Conference (SPLC'11), IEEE, Munich, Germany. pp. 1–4.
- [49] Erdweg, S., van der Storm, T., Völter, M., Boersma, M., Bosman, R., Cook, W.R., Gerrtsen, A., Hulshout, A., Kelly, S., Loh, A., Konat, G.D.P., Molina, P.J., Palatnik, M., Pohjonen, R., Schindler, E., Schindler, K., Solmi, R., Vergu, V.A., Visser, E., 2013. The State of the Art in Language Workbenches, in: Erwig, M., Paige, R.F., Van Wyk, E. (Eds.), Proceedings of the 6th International Conference on Software Language Engineering (SLE'13), Springer, Indianapolis, USA. pp. 197–217.
- [50] Favalli, L., Kühn, T., Cazzola, W., 2020. Neverlang and FeatureIDE Just Married: Integrated Language Product Line Development Environment, in: Collet, P., Nadi, S. (Eds.), Proceedings of the 24th International Software Product Line Conference (SPLC'20), ACM, Montréal, Canada. pp. 285–295.
- [51] Fenwick, P.M., 1994. A New Data Structure for Cumulative Frequency Tables. Software: Practice and Experience 24, 327–336.
- [52] Fowler, M., 2005. Inversion of Control. Martin Fowler's Blog. URL: <https://martinfowler.com/bliki/InversionOfControl.html>.
- [53] Frakes, W., Terry, C., 1996. Software Reuse: Metrics and Models. ACM Computing Surveys 28, 415–435.
- [54] Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Professional Computing Series, Addison-Wesley, Reading, Ma, USA.
- [55] Graham, S.L., Haley, C.B., Joy, W.N., 1979. Practical LR Error Recovery, in: Johnson, S.C. (Ed.), Proceedings of the 1979 Sigplan Symposium on Compiler Construction (CC'79), ACM, Denver, CO, USA. pp. 168–175.
- [56] Gray, II, E.J., 2007. TextMate: Power Editing for the Mac. Pragmatic Bookshelf.
- [57] Griss, M.L., 2000. Implementing Product-Line Features with Component Reuse, in: Frakes, W.B. (Ed.), Proceedings of the 6th International Conference on Software Reuse (ICSR'00), Springer, Vienna, Austria. pp. 137–151.
- [58] Grönninger, H., Krahn, H., Rumpe, B., Schindler, M., Völkel, S., 2008. MontiCore: A Framework for the Development of Textual Domain Specific Languages, in: Schäfer, W., Dwyer, M., Gruhn, V. (Eds.), Companion Proceedings of the 30th International Conference on Software Engineering (Companion ICSE'08), IEEE, Leipzig, Germany. pp. 925–926.



- [59] Gunasinghe, N., Marcus, N., 2022. Language Server Protocol and Implementation: Supporting Language-Smart Editing and Programming Tools. Apress.
- [60] Hartmann, H., Trew, T., 2008. Using Feature Diagrams with Context Variability to Model Multiple Product Lines for Software Supply Chains, in: Geppert, B. (Ed.), Proceedings of the 12th International Software Product Line Conference (SPLC'08), IEEE, Limerick, Ireland. pp. 12–21.
- [61] Haugen, Ø., Møller-Pedersen, B., Oldevik, J., Olsen, G.K., Svendsen, A., 2008. Adding Standardized Variability to Domain Specific Languages, in: Pohl, K., Geppert, B. (Eds.), Proceedings of the 12th International Software Product Line Conference (SPLC'08), IEEE, Limerick, Ireland. pp. 139–148.
- [62] Heidenreich, F., Johannes, J., Karol, S., Seifert, M., Wende, C., 2011. Model-Based Language Engineering with EMFText, in: Lämmel, R., Visser, J., Saraiva, J. (Eds.), Proceedings of the International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'11), Springer, Braga, Portugal. pp. 322–345.
- [63] Hindley, R., 1969. The Principal Type-Scheme of an Object in Combinatory Logic. Transactions of the American Mathematical Society 146, 29–60.
- [64] Holl, G., Grünbacher, P., Rabiser, R., 2012. A Systematic Review and an Expert Survey on Capabilities Supporting Multi-Product Lines. Information and Software Technology 54, 828–852.
- [65] Hotz, L., Wolter, K., Krebs, T., Deelstra, S., Sinnema, M., Nijhuis, G.J., MacGregor, J., 2006. Configuration in Industrial Product Families: The ConIPF Methodology. IOS Press.
- [66] Hu, H., Shinde, S., Adrian, S., Chua, Z.L., Saxena, P., Liang, Z., 2016. Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks, in: Shmatikov, V., Erlingsson, U. (Eds.), Proceedings of the 2016 IEEE Symposium on Security and Privacy (SP'16), IEEE, San José, CA, USA. pp. 969–986.
- [67] Hudak, P., 1998. Modular Domain Specific Languages and Tools, in: Devanbu, P., Poulin, J. (Eds.), Proceedings of the 5th International Conference on Software Reuse (ICSR'98), IEEE, Victoria, BC, Canada. pp. 134–142.
- [68] Hughes, J., 1989. Why Functional Programming Matters. The Computer Journal 32, 98–107.
- [69] Hürsch, W., Videira Lopes, C., 1995. Separation of Concerns. Technical Report NU-CCS-95-03. Northeastern University, Boston.
- [70] Jordan, T., Zib, S., 2024. A Langium-Based Approach to BigER. Bachelor's thesis. Technische Universität Wien. Wien, Austria.
- [71] Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S., 1990. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21. Carnegie Mellon University. Pittsburgh, Pennsylvania, USA.
- [72] Kästner, C., Apel, S., Ostermann, K., 2006. The Road to Feature Modularity?, in: Schäfer, I., John, I., Schmid, K. (Eds.), Proceedings of the 3rd Workshop on Feature-Oriented Software Development (FOSD'11), ACM, Munich, Germany.
- [73] Kats, L.C.L., Visser, E., 2010. The Spoofox Language Workbench: Rules for Declarative Specification of Languages and IDEs, in: Rinard, M., Sullivan, K.J., Steinberg, D.H. (Eds.), Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'10), ACM, Reno, Nevada, USA. pp. 444–463.
- [74] Kats, L.C.L., Visser, E., Wachsmuth, G., 2010. Pure and Declarative Syntax Definition: Paradise Lost and Regained, in: Proceedings of ACM Conference on New Ideas in Programming and Reflections on Software (Onward! 2010), ACM, Reno-Tahoe, Nevada, USA.
- [75] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, B., 2001. An Overview of AspectJ, in: Knudsen, J.L. (Ed.), Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01), Springer-Verlag, Budapest, Hungary. pp. 327–353.
- [76] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Videira Lopes, C., Loingtier, J.M., Irwin, J., 1997. Aspect-Oriented Programming, in: Akşit, M., Matsuoka, S. (Eds.), 11th European Conference on Object Oriented Programming (ECOOP'97), Springer-Verlag, Helsinki, Finland. pp. 220–242.
- [77] Klint, P., van der Storm, T., Vinju, J., 2009. EASY Meta-Programming with Rascal, in: Fernandes, J.M., Lämmel, R., Visser, J., Saraiva, J. (Eds.), Proceedings of the International Summer School on Generative and Transformational Techniques in Software Engineering III (GTTSE'09), Springer, Braga, Portugal. pp. 222–289.
- [78] Knuth, D.E., 1997. The Art of Computer Programming: Fundamental Algorithms. third ed., Addison Wesley.
- [79] Kosar, T., Bohra, S., Mernik, M., 2016. Domain Specific Languages: A Systematic Mapping Study. Information and Software Technology 71, 77–91.
- [80] Koscielny, J., Holthusen, S., Schaefer, I., Schulze, S., Bettini, L., Ferruccio, D., 2014. DeltaJ 1.5: Delta-Oriented Programming for Java 1.5, in: Childers, B. (Ed.), Proceedings of the 2014 International Conference on Principles

- and Practices of Programming on the Java platform: Virtual machines, Languages, and Tools (PPPJ'14), ACM, Cracow, Poland. pp. 63–74.
- [81] Krahn, H., Rumpe, B., Völkel, S., 2007. Efficient Editor Generation for Compositional DSLs in Eclipse, in: Tolvanen, J.P., Gray, J., Rossi, M., Sprinkle, J. (Eds.), Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling (DSM'07), Montréal, Canada.
- [82] Krahn, H., Rumpe, B., Völkel, S., 2010. MontiCore: A Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer* 12, 353–372.
- [83] Krueger, C.W., 1992. Software Reuse. *ACM Computing Surveys* 24, 131–183.
- [84] Krueger, C.W., 2006. New Methods in Software Product Line Practice. *Communications of the ACM* 49, 37–40.
- [85] Kühn, T., Cazzola, W., 2016. Apples and Oranges: Comparing Top-Down and Bottom-Up Language Product Lines, in: Rabiser, R., Xie, B. (Eds.), Proceedings of the 20th International Software Product Line Conference (SPLC'16), ACM, Beijing, China. pp. 50–59.
- [86] Kühn, T., Cazzola, W., Olivares, D.M., 2015. Choosy and Picky: Configuration of Language Product Lines, in: Botterweck, G., White, J. (Eds.), Proceedings of the 19th International Software Product Line Conference (SPLC'15), ACM, Nashville, TN, USA. pp. 71–80.
- [87] Kühn, T., Cazzola, W., Pirritano Giampietro, N., Poggi, M., 2019. Piggyback IDE Support for Language Product Lines, in: Thüm, T., Duchien, L. (Eds.), Proceedings of the 23rd International Software Product Line Conference (SPLC'19), ACM, Paris, France. pp. 131–142.
- [88] Kühn, T., Leuthäuser, M., Götz, S., Seidl, C., Abmann, U., 2014. A Metamodel Family for Role-Based Modeling and Programming Languages, in: Combemale, B., Pearce, D.J., Barais, O., Vinju, J. (Eds.), Proceedings of the 7th International Conference Software Language Engineering (SLE'14), Springer, Västerås, Sweden. pp. 141–160.
- [89] Kumar, A., Kumar, A., Iyyappan, M., 2016. Applying Separation of Concern for Developing Softwares Using Aspect Oriented Programming Concepts. *Procedia Computer Science* 85, 906–914.
- [90] Laddad, R., 2003. *AspectJ in Action: Pratical Aspect-Oriented Programming*. Manning Publications Company.
- [91] de Lara, J., Guerra, E., 2021. Language Family Engineering With Product Lines of Multi-Level Models. *Formal Aspects of Computing* 33, 1173–1208.
- [92] de Lara, J., Guerra, E., Bottoni, P., 2022. Modular Language Product Lines: A Graph Transformation Approach, in: Bencomo, N., Wimmer, M. (Eds.), Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems (MoDELS'22), ACM, Montréal, Canada. pp. 334–344.
- [93] Liebig, J., Daniel, R., Apel, S., 2013. Feature-Oriented Language Families: A Case Study, in: Collet, P., Schmid, K. (Eds.), Proceedings of the 7th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'13), ACM, Pisa, Italy.
- [94] van der Linden, F., Schmid, K., Rommes, E., 2007. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer.
- [95] Mäkitalo, N., Taivalsaari, A., Kiviluoto, A., Mikkonen, T., Capilla, R., 2020. On Opportunistic Software Reuse. *Computing* 102, 2385–2408.
- [96] Méndez-Acuña, D., Galindo, J.A., Combemale, B., Blouin, A., Baudry, B., 2016. Puzzle: A Tool for Analyzing and Extracting Specification Clones in DSLs, in: Kapitsaki, G.M., Santana de Almeida, E. (Eds.), Proceedings of the International Conference on Software Reuse (ICSR'16), Springer, Limassol, Cyprus. pp. 393–396.
- [97] Méndez-Acuña, D., Galindo, J.A., Combemale, B., Blouin, A., Baudry, B., 2017. Reverse Engineering Language Product Lines from Existing DSL Variants. *Journal of Systems and Software* 133, 145–158.
- [98] Méndez-Acuña, D., Galindo, J.A., Degueule, T., Combemale, B., Baudry, B., 2016. Leveraging Software Product Lines Engineering in the Development of External DSLs: A Systematic Literature Review. *Computer Languages, Systems & Structures* 46, 206–235.
- [99] Mernik, M., Heering, J., Sloane, A.M., 2005. When and How to Develop Domain Specific Languages. *ACM Computing Surveys* 37, 316–344.
- [100] Metzger, A., Pohl, K., 2014. Software Product Line Engineering and Variability Management: Achievements and Challenges, in: Dwyer, M.B., Herbsleb, J. (Eds.), Proceedings of Future of Software Engineering (FoSE'14), ACM, Hyderabad, India. pp. 70–84.
- [101] Milner, R., 1978. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences* 17, 348–375.
- [102] Mosses, P.D., 2004. Modular Structural Operational Semantics. *The Journal of Logic and Algebraic Programming* 60-61, 195–228.
- [103] Mosses, P.D., 2019. A Component-Based Formal Language Workbench, in: Monahan, R., Prevosto, V., Proença, J. (Eds.), Proceedings of the 5th Workshop on Formal Integrated Development Environment (F-IDE'19), Porto, Portugal. pp. 29–34.

- [104] Mosses, P.D., Vesely, F., 2014. FunKons: Component-Based Semantics in K, in: Escobar, S. (Ed.), Proceedings of the 10th International Workshop on Rewriting Logic and its Applications (WRLA'14), Springer, Grenoble, France. pp. 213–229.
- [105] Murphy, G.C., Kersten, M., Findlater, L., 2006. How Are Java Software Developers Using the Eclipse IDE? IEEE Software 23, 76–83.
- [106] Ng, K., Warren, M., Golde, P., Hejlberg, A., 2011. The Roslyn Project: Exposing the C# and VB Compiler's Code Analysis. White Paper. Microsoft.
- [107] Niephaus, F., Rein, P., Edding, J., Hering, J., König, B., Opahle, K., Scordialo, N., Hirschfeld, R., 2020. Example-Based Live Programming for Everyone: Building Language-Agnostic Tools for Live Programming with LSP and GraalVM, in: Proceedings of the ACM International Symposium on New Ideas, New Paradigms and Reflection on Programming and Software (Onward! '20), ACM, Virtual, USA. pp. 1–17.
- [108] van Ommering, R., 2001. Configuration Management in Component Based Product Populations, in: Westfechtel, B., Hoek, A. (Eds.), Proceedings of the International Workshop on Software Configuration Management (SCM'01), Springer, Toronto, Canada. pp. 16–23.
- [109] Pacak, A., Erdweg, S., Szabó, T., 2020. A Systematic Approach to Deriving Incremental Type Checkers, in: Grove, D. (Ed.), Proceedings of the 35th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'20), ACM, Chicago, IL, USA. pp. 1–28.
- [110] Parr, T., 2009. Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages. Pragmatic Bookshelf.
- [111] Pierce, B.C., 2002. Types and Programming Languages. MIT Press.
- [112] Pohl, K., Böckle, K., van der Linden, F.J., 2005. Software Product Line Engineering: Foundations, Principles and Techniques. Springer.
- [113] Prehofer, C., 1997. Feature-Oriented Programming: A Fresh Look at Objects, in: Akşit, M., Matsuoaka, S. (Eds.), Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97), Springer, Helsinki, Finland. pp. 419–443.
- [114] Prehofer, C., 2001. Feature-Oriented Programming: A New Way of Object Composition. Concurrency and Computation: Practice and Experience 13, 465–501.
- [115] Rabiser, R., Grünbacher, P., Dhungana, D., 2010. Requirements for Product Derivation Support: Results from a Systematic Literature Review and an Expert Survey. Journal of Information and Software Technology 52.
- [116] Rabiser, R., O'Leary, P., Richardson, I., 2011. Key Activities for Product Derivation in Software Product Lines. Journal of Systems and Software 84, 285–300.
- [117] Rask, J.K., Madsen, F.P., Battle, N., Macedo, H.D., Larsen, P.G., 2021. The Specification Language Server Protocol: A Proposal for Standardised LSP Extensions, in: Paskevich, A., Proença, J. (Eds.), Proceedings of the 6th Workshop on Formal Integrated Development Environment (F-IDE'21), Elsevier, Held On-Line. pp. 3–18.
- [118] Rentsch, T., 1982. Object Oriented Programming. Sigplan Notices 17, 51–57.
- [119] Richter, H., 1985. Noncorrecting Syntax Error Recovery. ACM Transactions on Programming Languages and Systems 7, 478–489.
- [120] Robinson, J.A., 1965. A Machine-Oriented Logic Based on the Resolution Principle. J. ACM 12, 23–41.
- [121] Rodríguez-Echeverría, R., Cánovas Izquierdo, J.L., Wimmer, M., Cabot, J., 2018a. An LSP Infrastructure to Build EMF Language Servers for Web-Deployable Model Editors, in: Hebig, R., Berger, T. (Eds.), Proceedings of the 2nd International Workshop on Model-Driven Engineering Tools (MDE-Tools'18), CEUR, Copenhagen, Denmark. pp. 1–10.
- [122] Rodríguez-Echeverría, R., Cánovas Izquierdo, J.L., Wimmer, M., Cabot, J., 2018b. Towards a Language Server Protocol Infrastructure for Graphical Modeling, in: Paige, R., Haugen, Ø. (Eds.), Proceedings of the 21st International Conference on Model Driven Engineering Languages and Systems (MoDELS'18), ACM, Copenhagen, Denmark. pp. 370–380.
- [123] Rosenmüller, M., Siegmund, N., 2010. Automating the Configuration of Multi Software Product Lines, in: David Benavides, D., Batory, D.S., Grünbacher, P. (Eds.), Proceedings of the 4th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS'10), Universität Duisburg-Essen, Linz, Austria. pp. 123–130.
- [124] Rosenmüller, M., Siegmund, N., Kästner, C., ur Rahman, S.S., 2008. Modeling Dependent Software Product Lines, in: Loughran, N., Groher, I., Lopez-Herrejon, R., Apel, S., Schwanninger, C. (Eds.), Proceedings of the GPCE Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering (McGPLe'08), University of Passau, Nashville, TN, USA. pp. 13–18.
- [125] Rosenmüller, M., Siegmund, N., Thüm, Saake, G., 2011. Multi-Dimensional Variability Modeling, in: Czarnecki, K., Eisenecker, U.W. (Eds.), Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems (VaMoS'11), ACM, Namur, Belgium. pp. 11–20.
- [126] Ryabko, B., 1992. A Fast On-Line Adaptive Code. IEEE Transactions on Information Theory 38, 1400–1404.



- [127] Schaefer, I., Bettini, L., Bono, V., Damiani, F., Tanzarella, N., 2010. Delta-Oriented Programming of Software Product Lines, in: Bosch, J., Lee, J. (Eds.), Proceedings of the 14th International Software Product Line Conference (SPLC'10), Springer, Jeju Island, South Korea. pp. 77–91.
- [128] Skiadas, C., Kjosmoen, T., 2007.  $\LaTeX$ ing with TextMate. The Prac $\TeX$  Journal 3.
- [129] Steinberg, D., Budinsky, D., Paternostro, M., Merks, E., 2008. EMF: Eclipse Modeling Framework. Addison-Wesley.
- [130] van der Storm, T., 2011. The Rascal Language Workbench. Technical Report SEN-1111. CWI.
- [131] Sweet, R.E., 1985. The Mesa Programming Environment. ACM Sigplan Notice 20, 216–229.
- [132] Vacchi, E., Cazzola, W., 2015. Neverlang: A Framework for Feature-Oriented Language Development. Computer Languages, Systems & Structures 43, 1–40. doi:10.1016/j.cl.2015.02.001.
- [133] Vacchi, E., Cazzola, W., Combemale, B., Acher, M., 2014a. Automating Variability Model Inference for Component-Based Language Implementations, in: Heymans, P., Rubin, J. (Eds.), Proceedings of the 18th International Software Product Line Conference (SPLC'14), ACM, Florence, Italy. pp. 167–176.
- [134] Vacchi, E., Cazzola, W., Pillay, S., Combemale, B., 2013. Variability Support in Domain-Specific Language Development, in: Erwig, M., Paige, R.F., Van Wyk, E. (Eds.), Proceedings of 6<sup>th</sup> International Conference on Software Language Engineering (SLE'13), Springer, Indianapolis, USA. pp. 76–95.
- [135] Vacchi, E., Olivares, D.M., Shaqiri, A., Cazzola, W., 2014b. Neverlang 2: A Framework for Modular Language Implementation, in: Proceedings of the 13th International Conference on Modularity (Modularity'14), ACM, Lugano, Switzerland. pp. 23–26.
- [136] Vanbrabant, R., 2008. Google Guice: Agile Lightweight Dependency Injection Framework. Apress.
- [137] Völter, M., 2011. Language and IDE Modularization and Composition with MPS, in: Lämmel, R., Saraiva, J.a., Visser, J. (Eds.), Proceedings of the 4th International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'11), Springer, Braga, Portugal. pp. 383–430.
- [138] Völter, M., Pech, V., 2012. Language Modularity with the MPS Language Workbench, in: Proceedings of the 34th International Conference on Software Engineering (ICSE'12), IEEE, Zürich, Switzerland. pp. 1449–1450.
- [139] Wachsmuth, G.H., Konat, G.D.P., Visser, E., 2014. Language Design with the Spoofox Language Workbench. IEEE Software 31, 35–43.
- [140] Wende, C., Thieme, N., Zschaler, S., 2009. A Role-Based Approach towards Modular Language Engineering, in: van den Brand, M., Gašević, D., Gray, J. (Eds.), Proceedings of the 2nd International Conference on Software Language Engineering (SLE'09), Springer, Denver, CO, USA. pp. 254–273.
- [141] White, J., Hill, J.H., Gray, J., Tambe, S., Gokhale, A., Schmidt, D.C., 2009. Improving Domain-specific Language Reuse with Software Product-Line Configuration Techniques. IEEE Software 26, 47–53.
- [142] Würthinger, T., Wimmer, C., Woß, A., Stadler, L., Duboscq, G., Humer, C., Richards, G., Simon, D., Wolczko, M., 2013. One VM to Rule Them All, in: Hirschfeld, R. (Ed.), Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward'13), ACM, Indianapolis, IN, USA. pp. 187–204.
- [143] Zschaler, S., Sánchez, P., Santos, J., Alférez, M., Rashid, A., Fuentes, L., Moreira, A., Araújo, J., Kulesza, U., 2009. VML\*—A Family of Languages for Variability Management in Software Product Lines, in: van den Brand, M., Gašević, D., Gray, J. (Eds.), Proceedings of the 2nd International Conference on Software Language Engineering (SLE'09), Springer, Denver, CO, USA. pp. 82–102.

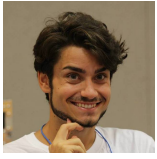


**Federico Bruzzone** is currently a Ph.D. student in Computer Science at Università degli Studi di Milano, Italy. He was born in 2000 and since he was a child he has been passionate about computer science and music. He got his bachelor degree in Musical Computer Science, the master degree in Computer Science and currently he is involved in the research activity of the ADAPT Lab. His main research interests are (but are not limited to) programming languages and compilers, software maintenance and evolution. For any question he can be contacted at [federico.bruzzone@unimi.it](mailto:federico.bruzzone@unimi.it).



**Walter Cazzola** is currently a Full Professor in the Department of Computer Science of the Università degli Studi di Milano, Italy and the Chair of the ADAPT laboratory. Dr. Cazzola designed the mChARM framework, @Java, [a]C#, Blueprint programming languages and he is currently involved in the designing and development of the Neverlang language workbench. He also designed the JavAdaptor dynamic software updating framework and its front-end FiGA. He has written over 100 scientific papers. His research interests include (but are not limited to) software maintenance, evolution and comprehension, programming methodologies and languages. He served on the program committees or editorial boards of the most important conferences and journals about his research topics. He is associate editor for the Journal of Computer Languages published by Elsevier. More information about Dr. Cazzola and all his

publications are available at <http://cazzola.di.unimi.it> and he can be contacted at [cazzola@di.unimi.it](mailto:cazzola@di.unimi.it) for any question.



**Luca Favalli** is currently a Computer Science Postdoctoral Researcher at Università degli Studi di Milano. He got his PhD in computer science from the Università degli Studi di Milano. He is involved in the research activity of the ADAPT Lab and in the development of the Neverlang language workbench and of

JavAdaptor. His main research interests are software design, software (and language) product lines and dynamic software updating with a focus on how they can be used to ease the learning of programming languages. He can be contacted at [favalli@di.unimi.it](mailto:favalli@di.unimi.it) for any question.