# Language Composition Untangled

Sebastian Erdweg, Paolo G. Giarrusso, and Tillmann Rendel

University of Marburg, Germany

**Abstract**

In language-oriented programming and modeling, software developers are largely concerned with the definition of domain-specific languages (DSLs) and their composition. While various implementation techniques and frameworks exist for defining DSLs, language composition has not obtained enough attention and is not well-enough understood. In particular, there is a lack of precise terminology for describing observations about language composition in theory and in existing language-development systems. To clarify the issue, we specify five forms of language composition: language extension, language restriction, language unification, self-extension, and extension composition. We illustrate this classification by various examples and apply it to discuss the performance of different language-development systems with respect to language composition. We hope that the terminology provided by our classification will enable more precise communication on language composition.

## 1 Introduction

Domain-specific languages (DSLs) are a prominent candidate for bridging the gap between domain concepts and software developers. DSLs enable software developers to think about the components and relations of a domain rather than about how these components and relations might be represented. DSLs thus provide abstraction over the concrete realization of domain concepts.

Not least due to the success of DSLs in practice, many language-development systems have been investigated [19]. To implement a DSL, a language developer can, for example, write a parser and interpreter, apply an attribute grammar system [9, 31], use a language workbench [7, 18], write a compiler plug-in for an extensible compiler [9, 20], or provide a library for domain primitives using regular functions [16], macros [29, 28], or sugar libraries [12]. Advances in DSL implementation techniques have led to a proliferation of DSLs in today's software engineering research and practice, and DSLs for many problem domains are available today.

However, realistic software projects are not just concerned with a single problem domain but also with many secondary domains such as data serialization and querying, communication, security, data visualization, graphical

user interfaces, concurrency, or logging. Following the idea of language-oriented software development [6, 36], we want to provide a separate DSL for each domain that occurs in a project and to use all of these DSLs together. Support for this large and changing amount of domains can only be efficiently provided if DSLs can be implemented independently and then composed together. Consequently, realistic software projects in a language-oriented context require *language composition*. Most recent work on language-development systems addresses language composition in one way or another.

At conceptual level, however, language composition is treated rather vaguely in the literature. In particular, there is no account the authors are aware of that specifies what language composition exactly means. This lack of a clear conceptual framework hinders our ability to reason about the composability of languages or to compare the support for language composition in different implementation techniques.

To this end, our goal is to provide precise terminology for language composition that enables effective communication on language composition and can serve as a basis for comparing existing and future language-development systems. In summary, we make the following contributions.

- We present a *classification of language composition* that distinguishes five cases: language extension, language restriction, language unification, self-extension, and extension composition. We illustrate this classification through various examples.
- We demonstrate that our classification provides precise terminology to explain language-composition support in existing technology and therefore clarifies our understanding of these systems.
- We apply our terminology to show that many language-development systems employ multiple forms of language composition. Without precise terminology, these different applications of language composition can easily be confused.
- Our classification reveals unexpected room for improvement for language-composition support in existing language-development systems. In fact, only one of the systems we investigated supports the composition of independent languages.

## 2 Language composition

The term "language composition" can refer to mechanisms and usage scenarios that significantly differ in terms of flexibility and reuse opportunities. In fact, the composability of languages is not a property of languages themselves: any two languages can be composed by stipulating a new syntax and semantics for the composed language. Rather, language composability is a property of language definitions, that is, whether two definitions work together without changing them.

To clarify the situation, we develop a taxonomy of language composition based on the idea of unchanged reuse, that is, whether a language definition can be reused without modifying it. Existing language-development systems differ significantly in their support for unchanged reuse. For example, some systems support the unchanged reuse of a base language through extension (e.g., macro systems), whereas other systems even allow to compose independently developed languages unchanged (e.g., JastAddJ). To avoid ambiguous statements, authors need to be aware of the equivocality of language composition and we recommend to consciously use language composition only as an umbrella term for our more precise terminology.

## 2.1 Language extension ($\lhd$)

When the first stable version of Java was released, it lacked many features that we are used to today. For example, before version 1.5, Java had no support for the foreach loop or generics. Java was only extended with these features later on. Similarly, earlier versions of Haskell did not include support for `let` expressions (introduced in Haskell 1.1), monads, or `do` notation (both introduced in Haskell 1.3) [17]. By now, these later-added features have become characteristic for Java and Haskell, respectively. More generally, languages evolve over time and subsequent introduction of language features is usual.

This brings us to the first form of language composition: *language extension*. A language designer composes a base language with a language extension. A language extension is itself a language fragment, which typically makes little sense when regarded independent of the base language. This dependency of the language extension on the base language is the main characteristic of this form of language composition.

Often, implementing a language extension involves changing the implementation of the base language. Examples include the integration of generics into Java and `do` notation into Haskell. However, the language-engineering community has brought forward language-development systems that particularly support language extensibility. These systems share a common property, which we capture in the following definition.

**Definition 1** *A language-development system supports language extension of a base language if the implementation of the base language can be reused unchanged to implement a language extension.*

Importantly, this definition only demands the reuse of the base language's implementation but does not regulate how language extensions are implemented. In particular, this definition does not prescribe whether multiple language extensions can be used jointly. In addition to describing terminology, we also introduce an algebraic notation for language composition. We will later use this notation to explain how different forms of language composition integrate. We denote the result of composing a base language $B$ with a language extension $E$ as $B \lhd E$. The asymmetry of the language-composition operator $\lhd$ reflects the dependency of the extension on the base language.

**Language restriction.** Especially in education, it sometimes makes sense to restrict an existing programming language. For example, to teach students functional programming in Haskell, monads and type classes are rather hindering. It might be more instructive to rigorously forbid the use these constructs. We call this language restriction as opposed to language extension.

Interestingly, language restriction does not require special support by language-development systems. Instead, a language restriction can be implemented as an extension of the validation phase of the base language: The extension rejects any program that uses restricted language constructs. The same idea is used in pluggable type systems [2]. Since language extension subsumes language restriction, we do not treat language restriction specifically in the remainder of this paper.

## 2.2 Language unification ($\uplus$)

Language extension and language restriction assume the existence of one dominant (typically general-purpose) language that can serve as the base of other languages. However, sometimes it is more natural to compose languages on equal terms. For example, consider the composition of HTML and JavaScript. Both languages serve a purpose and can be used independently: HTML for describing web pages and JavaScript as a prototype-based programming language. If anything, it would make sense to use the general-purpose language JavaScript as a base language for HTML. However, in the domain of dynamic web pages, the HTML-based view is the central program artifact.

Accordingly, we want to compose languages in an unbiased manner. Furthermore, the language composition should be deep and bidirectional, that is, program fragments from either language should be able to interact with program fragments from the other language. For example, in the composition of HTML and JavaScript as defined by the W3C [35], JavaScript programs can manipulate and generate HTML documents using the DOM tree or `document.write()`, and dynamic JavaScript-based behavior can be attached to HTML elements using attributes like `onMouseOver="showPopup()"`. In summary, to compose HTML and JavaScript, we need to add primitive support to JavaScript for generating HTML document trees and to supplement the definition of HTML elements to allow event attributes.

This illustrates the next form of language composition: *language unification*. A language designer composes two independent languages by unification. Like in mathematical unification, language unification requires that parts of the languages are equalized. For example, deep integration often requires sharing of primitive data types such as numbers or strings. Also, like in mathematical unification, the unified language subsumes its two constituents.

Language unification is very difficult to achieve in practice and rarely supported by language development systems. Often language unification requires the composition of language implementations by hand. The reason for this seemingly incompatibility of languages is the lack of a common back-end (for example, compiled for different VMs or separate interpreter engines). Unification

is simpler if the same language-development system implements both languages. In particular, sometimes support for language extension suffices to unify two languages, for example, regular expressions and Java. More generally, though, we apply the following definition.

**Definition 2** *A language-development system supports language unification of two languages if the implementation of both languages can be reused unchanged by adding glue code only.*

Notably, this definition permits the adaption of the unified languages as long as their implementations remain unchanged. Generally, we can assume that some program weaves the two language implementations together. As usual in component engineering and modularity discussions, we refer to the program that weaves two languages as glue code.

We write $L_1 \uplus_g L_2$ to denote the language that unifies $L_1$ and $L_2$ with glue code $g$. The symmetry of the language operator $\uplus$ reflects that unification composes languages on equal terms. Due to glue code, though, $\uplus$ is not necessarily a symmetric relation, that is, $L_1 \uplus_g L_2$ only equals $L_2 \uplus_g L_1$ for different glue code $g$. Moreover, the unification of two languages is typically not unique. For example, in HTML $\uplus_g$ JavaScript, the glue code $g$ determines the attribute name `onMouseOver`, which might as well be called `onPointerOver`.

## 2.3 Self-extension ($\hookleftarrow$)

For many subdomains of a software project exist special-purpose languages that provide functionality specific to the domain. Examples of such DSLs include SQL for data querying, XML for data serialization and regular expressions for string analysis. Since these languages each only tackle a small part of a software system, it makes sense to make their functionality available in a general-purpose language that can serve as a bridge between these DSLs.

Traditionally, this form of language composition is called language embedding: A domain-specific language is embedded into a host language by providing a host-language program that encapsulates the domain-specific concepts and functionality [16]. However, the term "language embedding" is ambiguous because it only describes the result of integrating one language into another language. However, such integration can not only be achieved with pure-embedding-like techniques but also using language extension in an extensible compiler, for example, where the embedding is described as a compiler plugin. Since the decisive difference to other forms of language composition is *how* we integrate languages, our terminology should reflect that. In particular, we aim to exclude systems where the extensibility is external to the host language.

We call this form of language composition *self-extension*. To compose a host language with an embedded language, a language implementor develops, in the host language itself, a program which defines the embedded language. Often the definition of the embedded language simply consists of a host-language API for accessing domain-specific concepts and functionality. More advanced languages

also enable the self-extension of the host language's syntax, static analyses, and IDE support. Because the implementation of an embedded language is itself a regular program of the host language, the host language extends itself.

There are various ways of self-extending a language, but two extension styles are most popular: string embedding and pure embedding. In string embedding, a program of the embedded language is represented as a string of the host language and the embedded language provides an API for evaluating embedded programs. A good example of string embedding is the integration of regular expressions into Java (similar for many other host languages). A programmer writes a regular expression `"a[b-z]*"` as a string and passes it to the library function `Pattern.match` as in `Pattern.match("a[b-z]*","atext")`. `Pattern.match` parses and compiles the regular expression at runtime and matches it against the given input text `"aText"`. Another example for string embedding is the integration of SQL into Java, where SQL queries are represented as Java strings (see package `java.sql`). Generally, string-embedded programs do not compose well with each other because string embedding reifies a lexical macro system [11]. In particular, string embeddings are vulnerable to injection attacks [3].

Alternatively, programs of the embedded language can also be expressed as a sequence of API calls in the host language. Paul Hudak coined the term pure embedding for this kind of self-extension [16]. As an example, consider the embedding of XML into Java using JDOM. A program of the embedded language XML is simply a Java program that utilizes the JDOM API:

```
Element book = new Element("book");
book.setAttribute("title", "Sweetness and Power");
Element author = new Element("author");
author.setAttribute("name", "Sidney W. Mintz");
book.addContent(author);
```

A purely embedded language does not provide its own syntax but instead reuses the syntax of the host language. Therefore, programs of a purely embedded language can be readily mixed with code from the host language, for example, to retrieve the author name from a database.

Clearly, the term self-extension can only apply to languages and not to language-development systems in general. Accordingly, we define:

**Definition 3** *A language supports self-extension if the language can be extended by programs of the language itself while reusing the language's implementation unchanged.*

Self-extension has two essential advantages over regular language extension. First, to run or compile a program of a self-extended host language, the standard interpreter or compiler of the host language is reused. In contrast, systems that support regular language extensions often require compiler configurations that reflect the activated extensions, which may differ for different source files. Second, since self-extensions are implemented in the self-extensible language itself, extensions can be used when writing further self-extensions. In particular,

this enables the integration of meta-DSLs, that is, DSLs for implementing further DSLs [12].

We write $H \leftarrow\!\!\shortmid E$ to denote the self-extension of a host language $H$ with the embedded language $E$. As defined above, the implementation of $E$ has to be an instance of $H$. The asymmetry of the language operator $\leftarrow\!\!\shortmid$ reflects this dependency of the embedded language on the host language.

## 2.4 Extension composition

So far, we have identified three language-composition scenarios a language-development system may support: language extension, language unification, and self-extension. However, these properties only describe to which extent a system supports base-language composition with a single extension or language. Our terminology so far does not describe to which extent a system supports the composition of extensions, that is, whether different extensions can work together.

Let us first note that systems which support language unification also support unification of extensions: $L \uplus_g (E_1 \uplus_h E_2)$. On the other hand, for systems that only support language extension, we need to distinguish three cases: no support for extension composition, support for incremental extension, and support for extension unification. In a system that does not support any form of extension composition, two extensions $B \lhd E_1$ and $B \lhd E_2$ cannot be used in combination at all. In contrast, in a system that supports incremental extension, an extended language $B \lhd E_1$ can in turn be extended to $(B \lhd E_1) \lhd E_2$. Here, extension $E_2$ may be specifically built to work on top of $E_1$. Incremental extension supports Steele's idea of growing a language [26]. Finally, in a system that supports extension unification, two independent extensions can be composed and used together $B \lhd (E_1 \uplus_g E_2)$ by using some glue code $g$. Extension unification supports growing a language modularly.

Self-extension adheres to the same case distinction for extension composability (no extension composability, incremental extension or extension unification). In addition, though, self-extensible languages support another interesting form of extension composition, namely self-application. Since implementations of extensions are programs of the host language itself, a host-language extension $E_1$ can be used in the implementation of another extension $E_2$, that is, $H \leftarrow\!\!\shortmid E_2$ where $E_2 \in (H \leftarrow\!\!\shortmid E_1)$.

This discussion shows that language composition is not only important for the base language but also for extensions. Therefore, precise terminology is crucial to enable clear statements about the language-composition support of a system and to prevent confusion about whether a statement addresses base-language composability or extension composability. Furthermore, this discussion illustrates the utility of an algebraic notation for describing and reasoning about language composition.

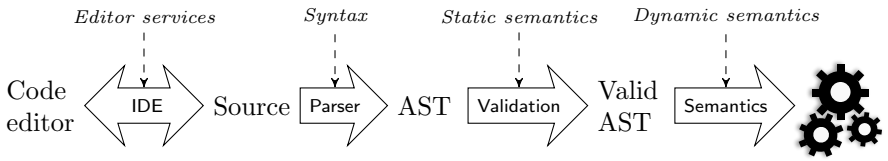


Figure 1: A typical language processing pipeline.

# 3 Language components

Support for language composition is often not uniform for all components of a language definition because different low-level techniques and high-level considerations apply to different aspects of a language. Generally, a language consists of syntax and semantics. Accordingly, most language definitions stipulate the syntax and semantics of a language separately. However, for machine-processed languages and programming languages in particular, this picture is not entirely correct. In fact, the definition of many machine-processed languages consists of three artifacts: a context-free syntax, a collection of non-context-free validation procedures (the static semantics), and a definition of the language's behavior (the dynamic semantics). While the reason for separating context-free syntax and validation is a technical one—generic context-sensitive parser frameworks are inefficient—we cannot ignore the implications on language design and language composition.

The relation between language-definition artifacts is depicted in Figure 1. First, a parser checks whether the input source code adheres to the given context-free grammar and either rejects the program with an error message or produces an abstract syntax tree. Subsequently, the language validation procedure processes the resulting syntax tree and either accepts or rejects it, together with the original source artifact. If the code is not valid, validation generates an error report. If the program is instead valid, validation may add information to the AST (for instance, overload resolution in Java). Next, the language's (dynamic) semantics takes a syntax tree as input and produces the meaning of the corresponding program. The behavior of the dynamic semantics may be unspecified for programs which are rejected during parsing or validation.

In addition to these classical components of a language processing pipeline, we include integrated development environments (IDEs) as a fourth component into Figure 1 and the discussion in the present paper. IDEs provide an editor with various editor services to the programmer. Editor services may include syntax coloring, code outline, code folding, code completion, reference resolving to jump to the definition of an identifier, or refactorings. More generally, this component includes all programming tools that a developer can use to write, navigate or maintain programs. While IDE support is not directly part of a language definition, it is essential for the productivity of programmers. Furthermore, only few systems exist that support the composition of IDE support for different

languages.

Our separation of languages into four components is general and covers any programming language. For instance, the Java programming language [13] declares a context-free syntax, a type checker, and a compiler that produces byte code. Instead of using a general context-sensitive parser to parse Java's context-sensitive syntax directly, compilers parse the context-free syntax first before applying special-purpose validations such as type checking and the remainder of compilation. In addition, various IDEs for Java exist, for example, Eclipse or IntelliJ IDEA. Another example language is XML: XML's context-free syntax and XML validity can both be checked efficiently, whereas the application of a general-purpose context-sensitive parser will likely lead to inefficient XML processing. Finally, note that language components similarly exist for DSLs such as SQL, VHDL, or DOT.

However, some languages combine two or more of the language components we identified. Prominently, dynamically typed languages such as Ruby or Smalltalk perform well-typedness validation as part of their dynamic semantics. Alternatively, type checking and parsing can be combined to resolve syntactic ambiguities by typing information [4]. LaTeX even applies parsing and validation as part of its dynamic semantics: it repeatedly parses, validates and executes the next command or macro until the complete source file is processed [11]. Finally, in Smalltalk, even the IDE is interpreted by the language's dynamic semantics and can be modified at runtime.

# 4 Existing technologies

We introduced new terminology for language composition in order to enable more precise descriptions of existing and future technologies. In this section, we exemplify the use of our terminology to classify existing language-development systems with respect to their language-composition support.

We reviewed existing language-development systems as described in the literature in light of our classification. Table 1 summarizes our findings. Each cell in the table shows how a system supports composition with respect to a specific language component, both regarding language extension or unification (first symbol) and regarding incremental extension or extension unification (second symbol, in parentheses). The last column applies to all language components and records whether a system supports self-extension. We have been somewhat liberal in our judgment for extension unification and also acknowledged support to systems that only support unification for non-interacting language extensions.

Different technologies follow very different approaches to achieve language composability. One of the simplest and also most popular mechanisms is hand-written preprocessors [25]. To extend a language, a programmer writes a preprocessor that translates the extended language into the base language. However, each extension requires its own preprocessor and preprocessors can only be composed sequentially, that is, run one after another. Consequently, preprocessors only support incremental extension but not extension unification.

| | Syntax | Validation | Semantics | IDE | Self-ext. |
|---|---|---|---|---|---|
| OpenJava [27] | | | $\triangleleft(\uplus)$ | | yes |
| pure embedding [16] | | $\triangleleft(\uplus)$ | $\triangleleft(\uplus)$ | | yes |
| MPS [34] | | $\triangleleft(\uplus)$ | $\triangleleft(\uplus)$ | $\triangleleft(\uplus)$ | yes |
| string embedding | $\triangleleft(\ )$ | | $\triangleleft(\ )$ | | yes |
| AspectLisa [22] | $\triangleleft(\ )$ | | $\triangleleft(\uplus)$ | | no |
| Converge [29] | $\triangleleft(\ )$ | $\triangleleft(\ )$ | $\triangleleft(\ )$ | | yes |
| preprocessors [25] | $\triangleleft(\triangleleft)$ | $\triangleleft(\triangleleft)$ | $\triangleleft(\triangleleft)$ | | no |
| Racket [28] | $\triangleleft(\triangleleft)$ | $\triangleleft(\uplus)$ | $\triangleleft(\uplus)$ | | yes |
| JSE [1] | $\triangleleft(\uplus)$ | | $\triangleleft(\uplus)$ | | yes |
| Helvetia [23] | $\triangleleft(\uplus)$ | | $\triangleleft(\uplus)$ | $\triangleleft(\uplus)$ | yes |
| ableJ [31] | $\triangleleft(\uplus)$ | $\triangleleft(\uplus)$ | $\triangleleft(\uplus)$ | | no |
| Polyglot [20] | $\triangleleft(\uplus)$ | $\triangleleft(\uplus)$ | $\triangleleft(\uplus)$ | | no |
| JastAddJ [9] | $\triangleleft(\uplus)$ | $\uplus(\uplus)$ | $\uplus(\uplus)$ | $\uplus(\uplus)$ | no |
| Spoofax [18] | $\uplus(\uplus)$ | $\triangleleft(\uplus)$ | $\triangleleft(\uplus)$ | $\triangleleft(\uplus)$ | no |
| SugarJ [12] | $\uplus(\uplus)$ | $\triangleleft(\uplus)$ | $\triangleleft(\uplus)$ | $\triangleleft(\uplus)$ | yes |

Table 1: Support for language composition in existing language-development systems: No composition (empty), extension but no extension composition $\triangleleft(\ )$, incremental extension $\triangleleft(\triangleleft)$, extension unification $\triangleleft(\uplus)$, language unification $\uplus(\uplus)$.

AspectLisa [22], ableJ [31] and JastAddJ[9] follow more sophisticated approaches and build on attribute grammars. Attribute grammars [8, 30] enable the definition of new productions to extend the base syntax and new attributes to extend the base language validation and semantics. Since AspectLisa and ableJ allow language extensions to reuse and extend base-language attributes, they support language extension, where the base language does not have to be changed. In addition, AspectLisa applies aspect-oriented programming to add new attributes to productions of the base language. On the other hand, JastAddJ applies aspect-oriented programming and rejects information hiding to support overwriting attributes. Accordingly, JastAddJ supports the composition of languages by unifying their respective implementations, that is, by only adding glue code and not changing previous implementations. The same also applies to IDE support [24].

Spoofax [18] follows an alternative approach to language composition based on SDF for syntax composition and Stratego for semantic composition. SDF [14] applies scannerless generalized LR parsing, which enables the unification of arbitrary context-free grammars. However, generalized parsing may result in a syntax tree that contains ambiguities. SDF supports the elimination of ambiguities on the basis of glue code, that is, without changing the original grammars. For semantic composition, Spoofax applies the Stratego term rewriting language [32], which supports adding rules to handle an extended base language. Stratego does

not support the adaption of an existing rule base, though, which is necessary to unify languages.

Polyglot [20] is an extensible compiler that allows language extensions to integrate into various compiler phases. For example, a language extension can extend the parsing, type checking, and code generation phase of the compiler to support additional language constructs. Polyglot achieves language extensibility with method delegation, where compiler actions are delegated to extensions, which further delegate to yet other extensions. Polyglot does not support language unification since adapting the behavior of extensions is not supported.

**Self-extensible languages.** The following language-development systems are self-extensible languages, that is, the base language itself is used to implement language extensions or glue code. The extended base language can then be used in the implementation of further self-extensions. Notwithstanding this similarity, self-extensible languages come in various flavors.

String embedding and pure embedding are approaches that apply to any base language that supports strings and code reuse. In string embedding, programmers use language extensions by writing specially-formatted strings of the base language, which the extension parses and evaluates at runtime of the program. A typical example of a string-embedded language is the language of regular expressions. The main problem of string embedding is the lack of proper structural abstraction. Therefore, string embeddings fall back to lexical abstraction and composition of program snippets, which is error-prone and forestalls static syntax analyses [11]. Furthermore, since IDEs require a structural representation of programs, string embedding comes without IDE support. Nevertheless, string embedding is widely applied in practice, for example, to issue SQL queries or generate XML documents.

Pure embedding takes a more structural approach than string embedding and represents programs as API calls [16]. In particular, a programmer can nest or sequentialize calls to such a special-purpose API. Moreover, API calls can readily be mixed with regular base language code as well as with calls to other special-purpose APIs. There is, however, one constraint that is often overlooked: Pure embeddings must share their data representations. For example, suppose an extension provides its own collection data type. This prevents reuse of functionality from the base language such as mapping or sorting as well as integration with other extensions that can only process standard collections. As pointed out by Mernik et al. [19], pure embedding enables the reuse of IDE support of the base languages such as code completion for a special-purpose API. However, true domain-specific editor services such as SQL-specific code coloring is not in the focus of pure embedding.

Converge [29], JSE [1], OpenJava [27], and Racket [28] enable language extensions with macros and macro-like facilities. A macro is much like a normal function except it is run at compile-time. Consequently, a macro does not take or produce normal runtime data, but instead takes and produces compile-time data, that is, representations of programs. Converge, JSE, and Racket represent

programs as syntax trees, whereas OpenJava represents programs as metaobjects. None of these systems support language unification since the meaning of a previously defined macro cannot be changed. However, some macro systems come with more advanced support for unifying independent language extensions. For example, Racket supports extension unification through local and partial macro expansion, which enables the collaboration of independent macros [28].

SugarJ [12] is similar to macro systems but supports more flexible syntax composition. Like Spoofax, SugarJ employs SDF [14] to support the unification of arbitrary context-free grammars, where additional glue code can coordinate between grammars to eliminate ambiguities. To specify the validation and semantics of extensions, SugarJ uses Stratego's support for composing partial pattern matches through equally-named rules. Since pattern matches can only be added, SugarJ does not support the unification of an extension's validation or semantics. Moreover, SugarJ provides IDE support for the base language and extensions [10]. IDE support is extensible because it aggregates information from all extensions (e.g., for code completion) or chooses the most specific editor service available (e.g., for syntax coloring).

Helvetia [23] leverages Smalltalk's dynamic nature to enable extensibility of parsing, compilation, and IDE support. Helvetia extensions are implemented through annotated methods, which Helvetia organizes in a global rule set. Whenever two or more rules are active in the parser, compiler, or IDE, Helvetia throws an error. It is not possible to adapt existing extensions non-invasively.

The projectional language workbench MPS [34] rejects parsing and applies intentional programming instead. Essentially, MPS maintains a central program representation, which can be thought of as an AST, and displays projections of the AST to the programmer. To edit a program, a programmer sends edit directives to MPS, which applies the edits to the central AST and updates the projection. This way MPS provides IDE support and creates a user experience close to usual programming environments. Furthermore, MPS supports extensibility: The central program representation can be extended by new concepts, which can integrate into existing projections, validations, and code generation. As in the other systems, once defined, the behavior of an extension is fixed [33].

**Summary.** We have shown how our terminology for language composition is useful to explain existing systems and distinguish between them meaningfully. In particular, our terminology enables the precise description of composition support with the base language in contrast to composition support for language extensions.

We are aware that our discussion of existing technologies is incomplete and many more systems deserve attention. In particular, we excluded any tools from this discussion that do not support semantic extensibility, because without semantics programs of an extended language cannot be run. However, since the goal of this work is the clarification of language composition in general, we believe the omission of any particular system is negligible. Furthermore, we excluded semantic IDE services like debugging or testing from the present

discussion. An investigation of the composability of such services remains future work.

One important conclusion of our study is the lack of wide-spread support for language unification in existing systems. In our study, JastAddJ is the only tool that supports language unification for semantics. Language unification requires that a system supports the adaption of independently implemented languages, for example, by glue code. In JastAddJ, the flexible adaption by glue code is based on aspect-oriented programming. This suggests that technologies that favor flexibility over modularity in the sense of information hiding [21] should be more thoroughly investigated as a foundation for language-development systems.

# 5 Related work

Other authors have described DSL-related patterns but with less focus on reusability of language implementations. Spinellis [25] describes and classifies patterns for DSL design and implementation. Mernik et al. extend Spinellis' work and present an extensive survey [19] that covers various aspects of DSL development methodologies: They identify different DSL development phases, discuss when DSL development is appropriate, and compare different implementation techniques for DSLs. Mernik et al. also survey language-development systems and mention the use of DSLs as metalanguages within such systems. Spinellis and Mernik et al. distinguish whether an existing language is restricted or extended with new elements. As explained in Section 2.1, we instead identify these scenarios and consider language restriction as an extension to the validation system. In addition, we distinguish language unification, self-extension, and extension composability.

Hofer et al. [15] distinguish hierarchical and peer language composition in the context of embedded DSLs. We can describe hierarchical language composition through $(H \lhd L_1) \lhd L_2$ and peer language composition through $H \lhd (L_1 \uplus_g L_2)$. Our notation thus covers these scenarios while supporting the description of language-composition scenarios in a uniform way.

# 6 Conclusions

The goal of this paper is two-fold. First, we want to raise awareness on the many meanings of language composition and on the consequent ambiguity in discussions on language composition. For this ambiguity, we believe the lack of precise terminology deserves major blame. Therefore, our second goal is the classification of language composition and the introduction of precise terminology to describe language composition. We hope that the terminology introduced in this paper can clarify future discussions and communication on language composition.

**Acknowledgements.** We thank Christian Kästner and Klaus Ostermann for fruitful discussions and the reviewers for helpful comments on this paper. This work is supported in part by the European Research Council, grant No. 203099.

# References

[1] J. Bachrach and K. Playford. The Java syntactic extender (JSE). In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 31–42. ACM, 2001.

[2] G. Bracha. Pluggable type systems. In *OOPSLA Workshop on Revival of Dynamic Languages*, 2004. Available at `http://bracha.org/pluggableTypesPosition.pdf`.

[3] M. Bravenboer, E. Dolstra, and E. Visser. Preventing injection attacks with syntax embeddings. *Science of Computer Programming*, 75(7):473–495, 2010.

[4] M. Bravenboer, R. Vermaas, J. J. Vinju, and E. Visser. Generalized type-based disambiguation of meta programs with concrete object syntax. In *Proceedings of Conference on Generative Programming and Component Engineering (GPCE)*, volume 3676 of *LNCS*, pages 157–172. Springer, 2005.

[5] M. Bravenboer and E. Visser. Concrete syntax for objects: Domain-specific language embedding and assimilation without restrictions. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 365–383. ACM, 2004.

[6] S. Dmitriev. Language oriented programming: The next programming paradigm. Available at `http://www.jetbrains.com/mps/docs/Language_Oriented_Programming.pdf`, 2004.

[7] S. Efftinge and M. Voelter. oAW xText: A framework for textual DSLs. In *Workshop on Modeling Symposium at Eclipse Summit*, 2006.

[8] T. Ekman and G. Hedin. Rewritable reference attributed grammars. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *LNCS*, pages 144–169. Springer, 2004.

[9] T. Ekman and G. Hedin. The JastAdd extensible Java compiler. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 1–18. ACM, 2007.

[10] S. Erdweg, L. C. L. Kats, T. Rendel, C. Kästner, K. Ostermann, and E. Visser. Growing a language environment with editor libraries. In *Proceedings of Conference on Generative Programming and Component Engineering (GPCE)*, pages 167–176. ACM, 2011.

[11] S. Erdweg and K. Ostermann. Featherweight TeX and parser correctness. In *Proceedings of Conference on Software Language Engineering (SLE)*, volume 6563 of *LNCS*, pages 397–416. Springer, 2010.

[12] S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. SugarJ: Library-based syntactic language extensibility. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 391–406. ACM, 2011.

[13] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java(TM) Language Specification, (3rd Edition)*. Addison-Wesley, 2005.

[14] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF – reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

[15] C. Hofer, K. Ostermann, T. Rendel, and A. Moors. Polymorphic embedding of DSLs. In *Proceedings of Conference on Generative Programming and Component Engineering (GPCE)*, pages 137–148. ACM, 2008.

[16] P. Hudak. Modular domain specific languages and tools. In *Proceedings of International Conference on Software Reuse (ICSR)*, pages 134–142. IEEE, 1998.

[17] P. Hudak, J. Hughes, S. P. Jones, and P. Wadler. A history of Haskell: Being lazy with class. In *Proceedings of Conference on History of Programming Languages (HOPL)*, pages 1–55. ACM, 2007.

[18] L. C. L. Kats and E. Visser. The Spoofax language workbench: Rules for declarative specification of languages and IDEs. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 444–463. ACM, 2010.

[19] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37:316–344, 2005.

[20] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for Java. In *Proceedings of Conference on Compiler Construction (CC)*, volume 2622 of *LNCS*, pages 138–152. Springer, 2003.

[21] K. Ostermann, P. G. Giarrusso, C. Kästner, and T. Rendel. Revisiting information hiding: Reflections on classical and nonclassical modularity. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, volume 6813 of *LNCS*, pages 155–178. Springer, 2011.

[22] D. Rebernak, M. Mernik, P. R. Henriques, and M. J. V. Pereira. AspectLISA: An aspect-oriented compiler construction system based on attribute grammars. *Electronic Notes in Theoretical Computer Science*, 164(2):37–53, 2006.

[23] L. Renggli, T. Gîrba, and O. Nierstrasz. Embedding languages without breaking tools. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, volume 6183 of *LNCS*, pages 380–404. Springer, 2010.

[24] E. Söderberg and G. Hedin. Building semantic editors using JastAdd: Tool demonstration. In *Proceedings of Workshop on Language Descriptions, Tools and Applications (LDTA)*, pages 1–6. ACM, 2011.

[25] D. Spinellis. Notable design patterns for domain-specific languages. *Journal of Systems and Software*, 56(1):91–99, 2001.

[26] G. L. Steele, Jr. Growing a language. *Higher-Order and Symbolic Computation*, 12(3):221–236, 1999.

[27] M. Tatsubori, S. Chiba, M.-O. Killijian, and K. Itano. OpenJava: A class-based macro system for Java. In *Proceedings of Workshop on Reflection and Software Engineering*, volume 1826 of *LNCS*, pages 117–133. Springer, 2000.

[28] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as libraries. In *Proceedings of Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2011.

[29] L. Tratt. Domain specific language implementation via compile-time meta-programming. *Transactions on Programming Languages and Systems (TOPLAS)*, 30(6):1–40, 2008.

[30] E. Van Wyk, D. Bodin, J. Gao, and L. Krishnan. Silver: An extensible attribute grammar system. *Science of Computer Programming*, 75(1-2):39–54, 2010.

[31] E. Van Wyk, L. Krishnan, D. Bodin, and A. Schwerdfeger. Attribute grammar-based language extensions for Java. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, volume 4609 of *LNCS*, pages 575–599. Springer, 2007.

[32] E. Visser. Stratego: A language for program transformation based on rewriting strategies. In *Proceedings of Conference on Rewriting Techniques and Applications (RTA)*, volume 2051 of *LNCS*, pages 357–362. Springer, 2001.

[33] M. Voelter. Language and IDE modularization, extension and composition with MPS. In *Pre-proceedings of Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE)*, pages 395–431, 2011.

[34] M. Voelter and K. Solomatov. Language modularization and composition with projectional language workbenches illustrated with MPS. `http://voelter.de/data/pub/VoelterSolomatov_SLE2010_LanguageModularizationAndCompositionLWBs.pdf`, 2010.

[35] W3C HTML Working Group. HTML 4.01 specification. Available at `http://www.w3.org/TR/html4/`, 1999.

[36] M. P. Ward. Language-oriented programming. *Software – Concepts and Tools*, 15:147–161, 1995.