A Framework for Extensible Languages

Sebastian Erdweg
TU Darmstadt, Germany

Felix Rieger
TU Darmstadt, Germany

Abstract

Extensible programming languages such as SugarJ or Racket enable programmers to introduce customary language features as extensions of the base language. Traditionally, systems that support language extensions are either (i) agnostic to the base language or (ii) only support a single base language. In this paper, we present a framework for language extensibility that turns a non-extensible language into an extensible language featuring library-based extensible syntax, extensible static analyses, and extensible editor support. To make a language extensible, our framework only requires knowledge of the base language's grammar, the syntax for import statements (which activate extensions), and how to compile base-language programs. We have evaluated the generality of our framework by instantiating it for Java, Haskell, Prolog, JavaScript, and System F_ω, and by studying existing module-system features and their support in our framework.

Categories and Subject Descriptors D.2.11 [*Software Architectures*]: Languages; D.3.2 [*Language Classifications*]: Extensible languages; D.3.3 [*Language Constructs and Features*]: Frameworks

Keywords Macros; syntactic extensibility; compiler framework; module system; SugarJ

1. Introduction

Extensible programming languages enable programmers to introduce customary language features as language extensions of the base language. This serves two main purposes. First, a programmer can define language extensions for language constructs that are missing in the base language, such as tuples and first-class functions in Java. Second, extensible programming languages serve as an excellent base for language embedding, because the syntax, static analysis, and sometimes even the editor support of the embedded language can be realized as a language extension of the base language. This way, extensible languages combine the simplicity, composability, and base-language integration of internal DSLs with the flexibility of external DSLs [7]. To be unambiguous, when referring to extensible languages in this paper, we mean programming languages that at least provide some form of syntactic abstraction. Examples of extensible languages include Racket [14], SugarJ [10], and OCaml with camlp4 [4].

GPCE '13, October 27–28, 2013, Indianapolis, Indiana, USA.

ACM 978-1-4503-2373-4/13/10. . . $15.00.
http://dx.doi.org/10.1145/2517208.2517210

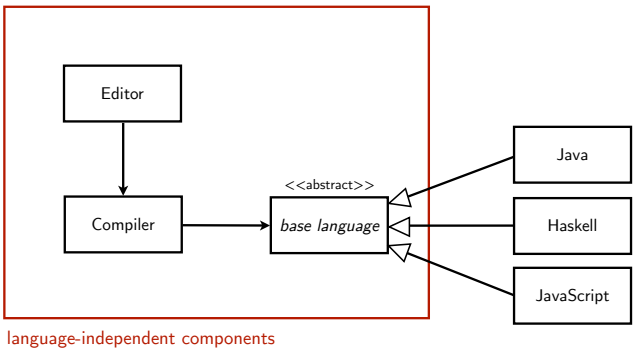

Figure 1. Architecture of the extensibility framework Sugar*.

In existing extensible languages, different techniques for syntactic abstraction have been applied. Most prominently, syntactic macros are compile-time functions that take syntactic objects as arguments and produce another syntactic object as output. Syntactic macros exist for many language, such as Scheme [5], Racket [15], C [31], Java [1, 26], Scala [2], Nemerle [24], or Haskell [23]. While the corresponding macro engines share many design decisions and implementation techniques, each system has its own implementation fully independent of the others. An alternative to syntactic macros are lexical macros, which take token sequences as input and produce another token sequence. Since lexical macros can be run before parsing takes place, the execution of lexical macros is agnostic to the base language. For example, the lexical macros of the C preprocesor (CPP) do, in fact, not depend on the C programming language and have been applied in many base languages including C++, Java, and Haskell. However, lexical macros are an unsatisfying mechanism for language extensibility since they cannot provide standard guarantees on the syntactic correctness of programs [9].

As an alternative to syntactic and lexical macros, in our prior work we proposed grammar-based language extensibility in the SugarJ programming language [10]. Instead of compile-time functions with explicit invocation syntax, SugarJ extensions extend the grammar of the base language with new productions that lead to extension-specific nodes in the abstract syntax tree (AST). In addition, a SugarJ language extension defines program transformations that transform the extension-specific parts of the AST into a base-language AST (the transformations may change base-language ASTs as well). Optionally, a SugarJ extension can define static analyses and editor services that SugarJ automatically executes. However, like existing syntactic macro systems, our implementation of SugarJ was specific to Java as a base language.

In this paper, we present Sugar*, a generalization of the SugarJ compiler as a framework for language extensibility that supports many different base languages with relatively little additional effort. To the best of our knowledge, Sugar* is the first reusable

implementation for extensible languages and syntactic abstraction. The basic architecture of our framework is illustrated in Figure 1. The core innovation is the abstract component base language that abstracts from concrete base languages. Essentially, the abstract base-language component abstracts from the following aspects of the base language: the grammar and pretty printer, the syntax for import statements (which activate extensions), and how to compile programs that result from an extension's program transformations. Since only these aspects have to be provided to the framework, the implementation effort to support a new base language is low. All extension-specific tasks are handled by the framework. We refactored the original SugarJ compiler such that it only depends on the abstract base-language component, but not on any specific base language such as Java or Haskell.

One important design decision that enabled the generalization of SugarJ into Sugar* was to use metalanguages independent of the base language. Specifically, we use SDF2 [29] for the declaration of syntax, Stratego [30] for the declaration of static analyses and program transformations, and Spoofax [19] for the declaration of editor services. We reuse these metalanguages for all base languages.

We have realized our framework for language extensibility in Java. The abstract base-language component is encoded as an abstract Java class whose abstract methods must be implemented for each concrete base language. To evaluate the generality of our framework, we have developed extensible languages based on Java, Haskell, Prolog, JavaScript, and System F_ω by instantiating the framework. We furthermore conducted a study of module-system features to identify the limits of our framework. In summary, we make the following contributions:

- We present the design of an abstract base-language component that can represent many different base languages, yet is detailed enough to realize syntactic extensibility on top of it.
- We present the design and implementation of a framework for language extensibility that provides extensibility support for syntax, static analyses, and editor support for a wide range of base languages.
- We instantiate the framework to realize extensible variants of Java, Haskell, Prolog, JavaScript, and System F_ω.
- We present a study of existing module-system features and discuss to which extent our framework supports these features.

2. Source-code processing with Sugar*

The Sugar* infrastructure distinguishes five separate phases: parsing, analysis, desugaring, generation, and editing. Figure 2 shows how these phases are connected in a pipeline to form a compiler and development environment. Sugar* applies this pipeline to a source file incrementally one toplevel declaration at a time, so that import statements (which activate language extensions) can affect the parser, etc. for the rest of the file. User extensions can customize all phases except for the final code-generation phase. In this section, we briefly describe the different phases and their respective responsibilities, and how users can apply customizations. This section is important to understand the interaction between the Sugar* processing and the base-language definition.

Parsing. The parser translates a textual source file into a structured syntax tree. We employ SDF2 and its scannerless GLR parser [29], which we have extended with support for layout-sensitive syntax [11] to enable base languages such as Haskell [12]. A user can customize parsing by leveraging SDF's support for grammar composition. In particular, it is possible to provide additional productions for a nonterminal of the base language. For example, the following SDF fragment extends the nonterminal JavaExpr from the base language Java:

```
context-free syntax
  XMLDoc -> JavaExpr {"XMLExpr"}
  XMLElem -> XMLDoc
  "<" Id Attr* "/>" -> XMLElem {"XMLEmptyElem"}
```

Sugar* composes user extensions with grammar of the base language and other extensions to obtain a parser for the composed language. This way it is possible to deeply intertwine syntax from different extensions in a single program. Accordingly, the Sugar* parser results in a syntax tree that contains nodes from the Sugar* base language and nodes specific to different extensions, such as XMLEmptyElem.

Technically, it is important that Sugar* performs incremental parsing: It parses a source file one toplevel declaration at a time. This is important because we want to change the current parser when encountering an import statement that refers to a language extension. When encountering such an import statement, we compose the imported grammar with the current grammar, regenerate a parser for the composed grammar (we use caching for performance), and continue with this parser.

Analysis. After parsing, Sugar* applies any static analyses that are defined as part of the base language. For example, for Java, we would define type checking as part of the base language. A Sugar* analysis receives the parsed syntax tree as input and is not allowed to change the structure of the parsed syntax tree; it may only add metadata to the syntax tree as annotations (illustrated by * in Figure 2). For example, the Java type checker would annotate types to expressions and variables in the syntax tree, but would not rewrite class references to fully qualified names. Such transformations can be realized in the next phase.

We use the strategic rewriting language Stratego [30] for implementing Sugar* analyses on top of syntax trees. Importantly, Stratego supports the composition of equally named rewrite rules that define alternative rewritings for some input. This allows Sugar* users to extend the analyses of the base language or to define additional ones specific to their language extension. Sugar* forwards the annotated syntax tree to the desugaring.

Desugaring. A desugaring implements the actual semantics of a user-defined language extension by translating programs of the extended syntax into programs of the base language. Like analyses, Sugar* desugarings are implemented in Stratego. The composition support of Stratego is essential for desugarings, because it allows Sugar* to compose desugarings of different extensions into a single one. Sugar* applies this composed desugaring bottom-up to the syntax tree until a fixed point is reached (rewriting strategy *innermost*). This way, Sugar* combines the semantics of different language extensions to form a single semantics for the composed extension.

Desugaring transformations can use the analysis information acquired in the previous phase (type-driven translation) and are free to translate any part of the syntax tree. This is unlike and more expressive than macros of most macro systems, which perform top-down expansion starting at the macro application. In particular, a Sugar* desugaring can also transform base-language programs, for example, to implement a custom optimization or to inject code for runtime monitoring (logging). However, whatever desugarings do, when a fixed point is reached, the resulting program may only consist of syntax-tree nodes of the Sugar* base language so that the subsequent generation phase can focus on the base language alone.

Generation. A Sugar* base-language syntax tree may describe a regular base-language program and/or a language extension. The generation phase receives such a tree and generates various artifacts

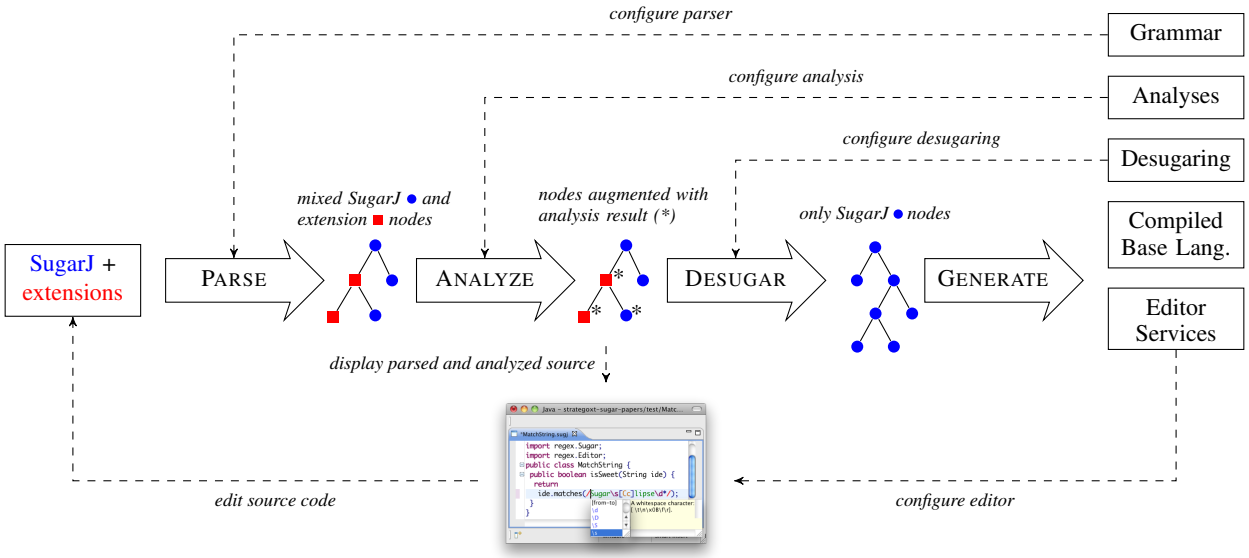

Figure 2. The SugarJ infrastructure: The result of processing is used to configure the parser, analysis, transformation, and editor.

depending on the nature of the syntax tree: If the syntax tree represents a base-language program, Sugar* calls the base-language compiler (if existent) to generate base-language binaries. For example, for Java, we call *javac* and write the corresponding *.class* files. For interpreted languages such as Prolog, we simply store the pretty-printed syntax tree for later execution. If the syntax tree represents a language extension, Sugar* generates separate artifacts for the different kind of extensions. If the extension defines an extended grammar, we generate a corresponding SDF module; if the extension defines an analysis, we generate a corresponding Stratego module; and so on. Since, at the time of writing, SDF and Stratego do not support separate compilation, we only generate pretty-printed artifacts and only call the respective compilers when we require the updated parser, analysis, or desugaring.

The generation phase is the only phase of Sugar* that is not customizable: The semantics of the base language is fixed once and for all in the definition of the base language (see next section), and the semantics of extensions is defined in terms of the base language. Thus, a customization of the generation phase is not required.

Editing. Language extensions break existing tools such as IDEs, which typically only support the unchanged base language. Sugar* builds on Spoofax [19] to automatically derive a simple Eclipse-based editor for the base language [8]. This editor can be further adapted to realize syntactic services (for example, syntax coloring, outline view, template-based code completion) and semantic services (for example, hover help or reference resolution) for the base language. Sugar* users can extend the editor support of the base language to accommodate, for example, extension-specific syntax coloring, hover help, or reference resolution. Sugar* editor services operate on the annotated syntax tree resulting from the analysis phase. Therefore, many editor services really are only concerned with the presentation of annotated information, such as showing type information in hover help or providing jump-to-definition support for resolved names.

Extension activation. Sugar* lifts a base language into an extensible language: Extensions of the lifted language can be defined *within* the lifted language itself as part of regular modules. For activating language extensions, Sugar* promotes the use of regular import statements. If an import statement refers to a module that declares a language extension, Sugar* adapts the current parser, analysis, desugaring, and editor by reconfiguring them. However, since imports are scoped, extensions are never activated globally but at most on a file-by-file basis.

The processing of the Sugar* framework is not independent of the base language. In the following section, we describe how the Sugar* processing interacts with the base language via an abstract representation.

3. An abstract representation of base languages

To support syntactic extensibility for a large number of programming languages, we define an abstract representation of programming languages. Our abstract representation captures the features needed for language extensibility, yet is generic enough to permit the instantiation with many different base languages. We split our abstraction into two parts: a base language and a base-language processor. The former is stateless and provides methods that reveal details about the base language in general, whereas the latter is stateful and provides methods for processing a single source file. We display our abstract representation for base languages and base-language processors in Figure 3.

3.1 Base language

The abstract base-language representation IBaseLanguage provides information per language and is independent of the processing of concrete source files. However, the base language gives rise to fresh base-language processors via the method createNewProcessor. The Sugar* compiler calls this method once for each source file it compiles.

File extensions. Sugar* distinguishes three kinds of files required respective file extensions from a base-language definition: Files that contain possibly extended base-language code and extension declarations (getSugarFileExtension), files that contain desugared, plain base-language code (getBaseFileExtension), and compiled base-language source files (getBinaryFileExtension). For the last one, a base language may return **null** to indicate that the language is interpreted and no compiled files exists. As example, our Java implementation returns `"sugj"`, `"java"`, and `"class"` as file extensions, respectively.

```
public interface IBaseLanguage {
  public IBaseProcessor createNewProcessor();
  public String getLanguageName();

  public String getSugarFileExtension();
  public String getBaseFileExtension();
  public String getBinaryFileExtension();

  public Path getInitGrammar();
  public String getInitGrammarModuleName();
  public List<Path> getPackagedGrammars();
  public Path getInitTrans();
  public String getInitTransModuleName();
  public Path getInitEditor();
  public String getInitEditorModuleName();

  public boolean isImportDecl(IStrategoTerm decl);
  public boolean isExtensionDecl(IStrategoTerm decl);
  public boolean isBaseDecl(IStrategoTerm decl);
}
```

```
public interface IBaseProcessor {
  public IBaseLanguage getLanguage();
  public void init(RelativePath sourceFile, Environment env);

  public void processModuleImport(IStrategoTerm toplevelDecl);
  public List<String> processBaseDecl(IStrategoTerm decl);

  public String getNamespace();
  public String getModulePathOfImport(IStrategoTerm decl);
  public boolean isModuleExternallyResolvable(String module);

  public String getExtensionName(IStrategoTerm decl);
  public IStrategoTerm getExtensionBody(IStrategoTerm decl);

  public Path getGeneratedSourceFile();
  public String getGeneratedSource();
  public List<Path> compile(List<Path> generatedSourceFiles,
                            Path targetDir,
                            List<Path> classpath);
}
```

Figure 3. Abstract representations for base languages (stateless) and their processors (stateful: one processor per source file).

Initialization. We make few assumptions about the structure of a language's programs and no assumptions about their syntax or tooling. Instead, a base language provides its own grammar, initial transformation (desugaring and analysis), and editor declaration. These provide the initial configurations for all customizable phases in Section 2: parsing, analysis, desugaring, and editing.

The initial grammar (getInitGrammar) must point to an SDF2 module, which typically requires other, pre-packaged SDF2 modules (getPackagedGrammars). In particular, the base-language grammar must define nonterminal ToplevelDeclaration, which Sugar* uses as the start symbol to parse the next toplevel declaration as explained in Section 2. The initial grammar must include productions for the declaration of syntactic extensions. To this end, the Sugar* standard library provides a pre-defined nonterminal ExtensionElem that can be integrated into the initial grammar. For example, the SugarJ grammar contains the following production in addition to standard Java:

```
JavaMod* "extension" JavaId "{" ExtensionElem* "}"
-> ToplevelDeclaration
```

The initial transformation (getInitTrans) must point to a Stratego module. Sugar* uses Stratego for analyses and desugarings, both of which can get initially defined here by implementing Stratego strategies start-analysis and desugar, respectively. In contrast to SDF2, Stratego modules do not occur pre-packaged, so that this additional method is only required for grammars.

Finally, the initial editor (getInitEditor) must point to a Spoofax editor-service module. This way, a base language can specify standard editor services such as syntax coloring or code completion. The user of the extensible language later can extend the initial grammar, transformation, and editor with custom rules.

AST predicates. Sugar* distinguishes only three kinds of toplevel declarations: import statements, extension declarations, and base-language declarations. These declarations have an abstract syntax that is specific to the base language. For example, import statements in Java look different from module-use statements in Prolog. Accordingly, we require the base language to provide predicates that allow us to distinguish imports, extensions, and base-language declarations. The AST predicates are used as the first step of the generation phase described in Section 2.

Importantly, the Sugar* compiler fully handles the processing and activation of extensions as well as the resolution and subcompilation of imported modules. For handling language-specific declarations such as a Java package, a Java class, a Haskell module header, or a Haskell module body, our compiler uses the base-language processor.

3.2 Base-language processor

The abstract base-language processor IBaseProcessor provides methods for source-file handling specific to the base language. A base-language processor is a stateful component that, in particular, is used to accumulate desugared source-file fragments during compilation of a sugared file. The Sugar* compiler acquires and uses exactly one base-language processor per source file and initializes it (init) with the path to the sugared source file and the compiler's environment. The environment contains common information such as the source path or the include path.

Base-language processing. In case the Sugar* compiler encounters a base-language declaration or an import statement that refers to another base-language module, the compiler requires the base-language processor to handle them (processModuleImport and processBaseDecl). Typically, these methods just pretty-print the abstract declaration term and store it until the source file is completely processed and the base-language compilation is triggered.

Our design permits a base-language declaration to establish further module dependencies. For example, a Java class can contain qualified names to that reference external classes or a Scala declaration can contain nested import statements. For this reason, processBaseDec yields a (possibly empty) list of additional module dependencies. Our compiler ensures that these additional dependencies are satisfied and, if needed, compiles the corresponding source files first.

Namespace. The Sugar* compiler requires base-language support for correctly treating the base-language's namespace. Generally, we assume that modules are organized in a hierarchical namespace that follows the file/directory structure. However, a base language can customize this behavior by providing non-standard implementations of getNamespace and getModulePathOfImport.

The compiler calls getNamespace to retrieve the namespace of the currently processed source file. In languages with hierarchical module systems that reflect the file/directory structure like Java or Haskell, getNamespace returns the directory path of the source file

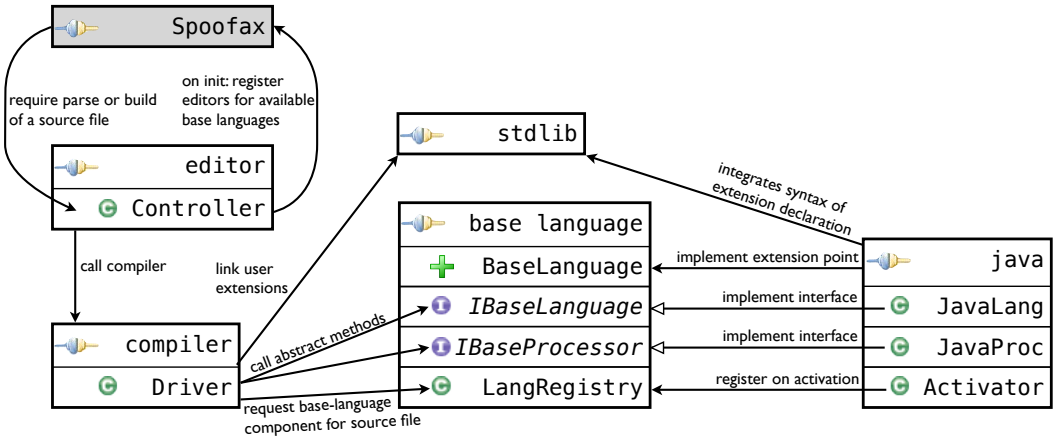

Figure 4. Detailed architecture of the extensibility framework Sugar*: OSGi modules, classes, interfaces, extension points.

relative to the root location of source files. In contrast, a language with a flat namespace returns the empty string or a constant string.

Conversely, to locate imported modules, the Sugar* compiler queries the base-language processor to retrieve the module path referred to by an import statement (getModulePathOfImport). Depending on the nature of the base-language module system, the returned path may reflect the hierarchical namespace. Our compiler will try to locate binaries, extensions, and source files of the returned module path. If this fails, the compiler would usually mark an illegal import statement. However, some languages like Haskell employ a package manager to resolve imported modules if no source/binary artifact exists. To allow for such module resolution, our compiler checks with the base-language processor if a module is externally resolvable (isModuleExternallyResolvable). We treat an import of an externally resolvable module just like an import of a pre-compiled base-language module.

Extensions. Sugar* handles all aspects of extensions independent of the base language. The only assistance the compiler needs, is to extract the name (getExtensionName) and the body (getExtensionBody) of an extension from its abstract-syntax representation, which may be base-language specific. For example, language extensions in Java occur in their own **public extension** declaration, whereas SugarHaskell extensions are regular modules containing grammar, transformation, and editor artifacts.

Generation. After processing all toplevel declarations of a source file, the Sugar* compiler queries the base-language processor for the generated base-language source code as a string (getGeneratedSource) and the path of a file into which the generated source code should be written. Typically, a base-language processor simply returns a concatenation of the pretty-printed, desugared toplevel declarations.

Finally, a base-language processor must define a method for compiling base-language source files (compile). This method receives a list of base-language source files that should be compiled, a target directory for the compiler output, and a classpath. The method receives a list of source files because the Sugar* compiler automatically detects cyclic imports. For each cycle, the compile method is only called once with all modules of the cycle as argument. This way, the base-language compiler can treat the cycle appropriately, for example, by rejecting it.

4. Technical realization of Sugar*

To realize Sugar*, we significantly reengineered the original SugarJ compiler at different levels. First, we introduced the abstract base-language component from the previous section as a level of indirection to parameterize the compiler over different base languages. Second, we used OSGi [21] to impose a large-scale module structure that separates the compiler, the editor, and the different base languages into separate but interdependent modules. Finally, we employed Eclipse extension points [17] to realize a central language registry that eliminates the dependency from the compiler to the concrete base languages. The final architecture is shown in Figure 4, which is a detailed version of the overview shown in Figure 1.

Parameterizing the compiler. The original SugarJ compiler incorporated specific knowledge about Java in various places. For example, it required a Java grammar to build the initial SugarJ grammar, it did a case distinction over different kinds of Java AST nodes, and it called the Java compiler to compile desugared programs. At all such places, we changed the implementation to call a corresponding method from the abstract base-language component BaseLang instead of using the Java-specific code. Simultaneously, we moved the old Java-specific code into a class JavaLang that instantiates BaseLang. After refactoring, we can use the refactored compiler and the extracted language component JavaLang to compile SugarJ files by instantiating the compiler with JavaLang: Driver.compile(**new** JavaLang(), sourceFile).

Imposing modules. Our framework should permit adding additional base languages independent of the compiler implementation. To this end, we decomposed the different artifacts of Sugar* into OSGi modules as follows:

editor: Operates as a bridge between the compiler of Sugar* and the Spoofax Eclipse plugin. In particular, registers the file-name extensions of available Sugar* languages with Spoofax and receives corresponding parse requests and compilation requests.

compiler: Contains classes that realize the processing explained in Section 2. In particular, the compiler contains code to build and execute the user-defined parsers and desugarings. Since the compiler has no dependency on the editor, it is possible to apply the Sugar* compiler as a batch compiler without a supporting editor.

stdlib: Contains the grammars of the metalanguages used by Sugar*: SDF2, Stratego, and Spoofax editor services. For defin-

SugarJ

```
package test;
import foo.Foo;
public sugar Bar {
  syntax
    A -> B {"Cons"}
}
```

SugarHaskell

```
module test.Bar
import foo.Foo

syntax
  A -> B {"Cons"}
```

SugarProlog

```
:- sugar_module(test/Bar).
:- use_module(foo/Foo).

:- syntax
     A -> B {"Cons"} .
```

SugarJS

```
module test/Bar
import foo/Foo
sugar {
  syntax
    A -> B {"Cons"}
}
```

SugarFomega

```
module test.Bar
import foo.Foo

syntax
  A -> B {"Cons"}
```

Figure 5. Extension declarations in different instantiations of Sugar*.

ing an extensible base language, developers mix these metalanguages with their base-language grammar. The standard library furthermore defines some commonly used Stratego operations, such as functions for performing analysis and annotating the result in a syntax tree. The compiler links extensions defined by the user against the standard library.

base language: Contains the abstract base-language component and the language registry. This module is used by the compiler and refined by concrete base languages.

concrete base language (java, etc.): Extends the abstract base-language component and, in particular, defines an initial grammar, an initial desugaring, and initial editor services for the extensible variant of the base language.

Language registry. OSGi employs a lazy activation policy of modules to minimize the number of loaded modules: A module is only loaded when the first class of this module is required [21]. Since the compiler of Sugar* should be independent of concrete base-language implementations, the compiler does not refer to any class from a base-language module. Thus, no base-language module would ever be activated by OSGi.

To circumvent this problem, we define an Eclipse *extension point* [17] and use Eclipse's *buddy policy* [17] to activate all concrete base-language modules whenever the abstract base-language module is activated. On activation, concrete base-language modules register themselves with the language registry. The compiler queries this language registry to receive a base-language component. This way, we achieve the decoupling of compiler and base languages:

1. The compiler can process any source file for which a base-language module is available.
2. The compiler is defined fully independently of any concrete base-language module.

5. Evaluation: SugarJ, SugarHaskell, SugarProlog, SugarJS, SugarFomega

To evaluate the generality of Sugar*, we instantiated the Sugar* framework by developing 5 extensible languages based on Java [10], Haskell [12], Prolog [22], JavaScript[1], and System F_ω [20]. Despite the significant difference between these languages regarding their syntax, semantics, and building, Sugar* successfully accommodates each of them. More yet, the design of Sugar* provides sufficient freedom for base languages to allow the integration of extensibility in a way that feels native to the language. For example, in Figure 5 we illustrate extension declarations in all 5 extensible languages we developed. Notably, for Java, Haskell, and Prolog we use standard-like module headers and import statements, and integrate extensibility declarations in a natural way. SugarJS and SugarFomega were implemented by others to provide extensibility for JavaScript and System F_ω, respectively. Since JavaScript and System F_ω do not have a standard module system, the respective developers designed a module system themselves. For example, the module system of SugarFomega syntactically resembles the module system of Haskell.

To realize an extensible variant of a base language, the interfaces presented in Section 3 had to be implemented. Specifically, for each base language, we defined or reused the following artifacts:

- A grammar of the pure base language specified with SDF2 [29].
- An SDF2 grammar module that integrates extension declarations (nonterminal ExtensionElem) into the base language and defines nonterminal ToplevelDeclaration for the base language. This module can serve as initial grammar.
- A Spoofax module that defines the initial editor services for the base language.
- An instance of IBaseLanguage.
- An instance of IBaseProcessor.

Except for Java, we did not define any initial transformations.

To illustrate the implementation of an extensible language with Sugar*, we present the relevant details of the SugarHaskell implementation. The SugarHaskell grammar is defined as follows:

```
module org/sugarj/languages/SugarHaskell
imports org/sugarj/languages/Haskell
        org/sugarj/languages/Sugar
exports context-free syntax
  ModuleDec          -> ToplevelDeclaration
  OffsideImportdecl  -> ToplevelDeclaration
  OffsideTopdeclList -> ToplevelDeclaration {cons("HSBody")}
  ExtensionElem+     -> ToplevelDeclaration {cons("ExtBody")}
```

This grammar defines the ToplevelDeclaration nonterminal by forwarding existing definitions for Haskell modules, Haskell imports, and Haskell toplevel declarations from the Haskell grammar org/sugarj/languages/Haskell and extension declarations from the grammar org/sugarj/languages/Sugar, that comes with the standard library of Sugar*. In Figure 6, we sketch the implementation of IBaseLanguage and IBaseProcessor for SugarHaskell using pseudo code.

The implementation shows that HaskellLanguage is stateless and simply functions as a body of knowledge about SugarHaskell. In contrast, HaskellProcessor stores the namespace and module name of the currently processed source file and accumulates the desugared source code when processModuleImport or processBaseDecl is called. For this, HaskellProcessor uses a pretty printer, which we generated from the Haskell base grammar (not shown). To check for externally resolvable modules, we call on the GHC packet manager ghc-pkg. Finally, compile runs the standard GHC compiler on the list of desugared files.

Implementation effort. As the implementation in Figure 6 indicates, the implementation effort for realizing an extensible variant of base language is modest using our framework. In Table 1, we summarize the source lines of code (SLOC: excluding empty

[1] https://github.com/bobd91/sugarjs/

```
public class HaskellLanguage implements IBaseLanguage {
  name = "Haskell"
  sugarFileExtension = "shs"
  baseFileExtension = "hs"
  binaryFileExtension = "o"
  initGram = Path("org/sugarj/languages/SugarHaskell.sdf")
  initTrans = Path("org/sugarj/languages/SugarHaskell.str")
  initEditor = Path("org/sugar/languages/SugarHaskell.serv")
  isImportDecl(x) = x.nodeName == "Import"
  isExtensionDecl(x) = x.nodeName == "ExtBody"
  isBaseDecl(x) = x.nodeName == "ModuleDec"
                || x.nodeName == "HSBody"
}
```

```
public class HaskellProcessor implements IBaseProcessor {
  var namespace; var moduleName; var source = ""
  processModuleImport(t) = source += prettyPrint(t)
  processBaseDecl(t) =
    source += prettyPrint(t)
    if t.nodeName == "ModuleDec"
      (namespace, moduleName) = splitName(prettyPrint(t))
    return [ ]
  modulePathofImport(t) = prettyPrint(t.subterm(1))
  isExternallyResolvable(s) = exec "ghc-pkg find-module $s"
  extensionName(t) = moduleName
  extensionBody(t) = t.subterm(1)
  genSourceFile = Path("$BIN/$namespace/$moduleName.hs")
  genSource = source
  compile(files, target, cp) =
    exec "ghc -outputdir $target -i $cp $files"
}
```

Figure 6. Instantiation of Sugar* for SugarHaskell.

	Base Grammar	Initial Grammar	IBase Language	IBase Processor
Java	1164	52	113	182
Haskell	923	10	92	168
Prolog	266	26	93	140
JavaScript	542	39	88	149
System F_ω	163	39	94	123

Table 1. SLOC for realizing extensible languages with Sugar*.

Feature	Base	Extensions
Flat namespace	●	●
Hierarchical namespace	●	●
Nested modules	●	○
First-class modules	●	○
External module management	●	○
Lexical imports	●	n/a
Qualified names	●	○
Selective import/renaming	●	○
Module reexport	●	◒
Nested imports	●	○
Cyclic imports	●	○
Dynamic module loading	●	○
Global compilation	◒	○
Incremental compilation	●	●
Separate compilation	◒	○
Interpreted	●	n/a

Table 2. Module system features and their support in Sugar*.

lines and comments) that were necessary to realize the different languages. As the table shows, the implementation effort per language is low. For all languages we considered, most effort actually had to be spent in developing a grammar for the base language. In particular for syntactically more complex languages like Java, Haskell, or JavaScript, developing a grammar for the base language is comparably laborious. The implementation of a new base language involves additional artifacts, such as constructor signatures, a pretty-printer, and static analyses for the base language.

This data suggests that the abstraction we designed for representing base languages is adequate in the sense that the instantiation is straight-forward and does not require involved coding. Especially, if one considers the boilerplate imposed by Java even for implementing simple functions, as opposed to the more concise pseudo-code implementation of Figure 6. Note that the numbers shown in Table 1 do not include the definition of a pretty-printer, which can be generically derived from the base-language grammar [3, 28]. Furthermore, we did not count the effort for developing static analyses of the base language since running these analyses inside Sugar* is optional and can be left to the base-language compiler. However, a reimplementation of base-language analyses in Stratego is required if they should be extended inside Sugar*.

In summary, Sugar* enables a full-fledged extensible language with little effort. The extensible language supports extensible syntax, extensible static analyses, extensible desugarings, and an extensible IDE. Moreover, even without any actual language extension, a base language realized with Sugar* already benefits from the dependency management of the Sugar* compiler and from the Spoofax-based Eclipse plugin that we provide [8, 19]. To show the generality of Sugar*, we successfully instantiated Sugar* with 5 base languages that employ diverse module-system features. In the subsequent section, we present a more general study of module-system features and their support in Sugar*.

6. A study of module-system features

We developed a framework for adding syntactic extensibility to existing programming languages. In our design of Sugar*, our main goal was generality: We want to support as many base languages as possible. Since Sugar* is largely module-driven (modules encapsulate extensions and extensions are activated through import statements), a deciding factor in the generality of Sugar* is whether it is possible to encode the module system of the base language as implementations of the interfaces IBaseLanguage and IBaseProcessor shown in Section 3. To better understand the generality of Sugar*, we investigate a subset of the module-system features of mainstream programming languages and discuss their support in Sugar*.

Table 2 gives an overview of the module-system features we studied. We distinguish whether Sugar* supports a module-system feature for base-language modules and whether it supports a feature for modules containing extension declarations. In the following, we describe the studied module-system features, name example languages that support the features, and discuss their support in Sugar* in detail.

Namespace (Flat: Prolog; Hierarchical: Haskell, Java): A module system's namespace can be either flat or hierarchical. In a flat namespace, modules only have a name that identifies them. In a hierarchical namespace, modules are organized in a hierarchical structure and are identified by their name and their path through this hierarchy. Sugar* was originally designed to support a base language with hierarchical namespaces (Java) for both language and extension modules. In our generalization, we retained

this feature through method getModulePathOfImport of interface IBaseProcessor. This method takes the syntax tree of an import statement and returns a relative *path* to the referenced module. A flat namespace can be modeled as hierarchical namespaces with an empty path prefix.

Nested modules (Java, Scala): Nested modules are submodules of a module which are defined in a module's body itself. Sugar* supports nested base-language modules as they do not expose any dependency to additional source files and can be fully handled by processBaseDecl. Since a nested module may be compiled to a separate binary (as is the case in Java and Scala), method compile returns a list of all generated files, which enables Sugar* to keep track of them and initiate a recompilation when necessary. However, we do not support extension declarations as nested modules, since we have no means of extracting the extension declaration from the outer module. Thus, extensions can only be declared as toplevel declarations.

First-class modules (Python, ML, Newspeak): First-class modules are modules that can be created dynamically, manipulated, and passed around as first-class values of the language. Similar to nested modules, we support first-class base-language modules, since they can be fully handled by processBaseDecl. Extensions cannot be declared in first-class modules since we cannot extract the extension declaration and we require extension declarations statically for parsing source code that uses the extension.

External module management (Haskell): External module management is a feature that some programming languages support to load pre-installed modules from external locations. An external location in this sense is any location outside the sourcepath and classpath used for compilation. For example, the Glasgow Haskell Compiler (ghc) looks up pre-installed modules in a package manager. We support external module management for base modules through method isModuleExternallyResolvable. However, externally resolved modules cannot define extension declarations, as we need to actually load and process extension declarations but we have no way of requesting the extension declaration from the external location.

Lexical imports (C, C++, Ruby): A lexical import inserts the source code of the imported module literally into the source file where the import occurs. The method processModuleImport in our interface is concerned with handling module imports. Implementations of a base language can choose to implement lexical import behavior for the base language, since we do not require that an import statement itself occurs in the final generated source code provided by getGeneratedSource. Lexical imports are not applicable for extensions, since an import of an extension declaration has a fixed semantics in Sugar*, namely to activate the extension in the scope of the importing module.

Qualified names (Java, Scala): Qualified names allow using members of a module without explicitly importing the containing module. For example, in Java this is accomplished by providing the hierarchical path to the module and the name of the member. Sugar* supports qualified names for base modules even though this entails dependencies to additional source files. To this end, method processBaseDecl returns a list of paths to additional module dependencies that occurred in the base declaration. For Java, processBaseDecl returns a list of all modules accessed through qualified names in the base declaration. We do not support qualified names for extension modules, as dependencies to extension declarations have to be explicit via import statements.

Selective import/renaming (Haskell, Prolog, Scala): A selective import allows to select which members are imported from a module. Renaming allows such members to be locally renamed. We support selective imports and renaming for base language modules, because we are only interested in the module dependency and allow method getModulePathOfImport to extract the path to the module from arbitrarily complex import statements. The selection and renaming of module members can be either realized by method processBaseDecl or simply forwarded to the base-language compiler. We do not support import selection/renaming for extension modules. Importing an extension will always bring the full extension into scope. We plan to investigate more fine-grained extension activation in our future work.

Module reexport (Haskell, Prolog)***:*** A reexport statement allows to export module members that have been imported from other modules. This way a module can collect and package functionality from multiple modules into a single module. We support module reexports for base-language modules, which can be handled as a standard base-language declaration by method processBaseDecl. We do not support customizable reexport of extension modules. Instead, an extension is never reexported by a base-language module, and an extension is always reexported by an extension module. Technically, the latter is due to transitive imports in the metalanguages SDF, Stratego, and Spoofax editor specifications, which underlie Sugar*.

Nested imports (Scala): A nested import is an import that occurs nested inside a code block. Specifically, a nested import is not a toplevel declaration. While Sugar* has special support for handling toplevel import statements, we also support nested imports for base-language modules. In principle, these nested imports can be handled by method processBaseDecl. However, since we want to keep track of module dependencies, we require processBaseDecl to provide a list of additional module dependencies, so that Sugar* can ensure these dependencies are resolvable and can initiate the compilation of the required modules. We do not support nested imports for extensions, when method processBaseDecl is called, the base declaration has already been parsed, analyzed, and desugared. Hence, it is too late for activating any language extension.

Cyclic imports (Java): Cyclic imports occur when two or more modules require features from each other so that a cyclic dependency graph is imposed. Cyclic imports are relevant for the Sugar* compiler because they essentially prevent incremental or separate compilation: In order to compile modules with cyclic dependencies, all involved modules have to be processed simultaneously. Sugar* supports cyclic dependencies of base-language modules by detecting cyclic dependencies and forwarding minimal strongly connected components to the method compile of interface IBaseProcessor. We do not support cyclic dependencies for language extensions, because the compilation of these extensions would depend on themselves, which is a circular definition.

Dynamic module loading (Python, Java): Languages that support dynamic module loading provide facilities for loading a module at runtime of a program. Accordingly, these module dependencies are not resolved statically at Sugar* compile time, but when the compiled program is executed. For example, Python resolves imports at runtime, which can be used to realize conditional import depending on some runtime computation. In Java, a class loader enables the dynamic loading of pre-compiled modules at runtime. We support base languages that feature dynamic loading of modules, because they do not influence compilation and no dependency tracking is necessary: The binary of a module does not change when a dynamically loaded module changes. We do not support dynamic loading of extension modules, as an extension influences the static parts of language processing: parsing, static analysis, desugaring, and editor support.

Global compilation (Stratego): Global compilation simultaneously processes all modules of a program. For example, Stratego collects all source modules and weaves them into a single module before continuing code generation. Sugar* partially supports global compilation of base-language source modules, because we do not enforce the compilation of single modules. However, Sugar* lacks a mechanism for adding a global-compilation phase after all modules have been processed. We will investigate better support for global compilation as described below for separate compilation. For language extensions, we do not support global compilation, since extensions have to be compiled before they can be used.

Incremental compilation (Java, Haskell): Incremental compilation first processes all imported modules before processing the a module itself. Sugar* supports incremental compilation for both base-language modules and extensions. In particular, Sugar* keeps track of module dependencies to initiate recompilation whenever a required module changes.

Separate compilation (C): Separate compilation processes source modules independently from each other by only relying on the interfaces of required modules. For example, a C source file typically does not import any other C source files but only header files from other modules. The C compiler compiles each source module independently and links the resulting binaries. Sugar* partially supports separate compilation, because method compile is free to compile modules that do not require other source modules. However, like for global compilation, Sugar* currently lacks a post-processing phase that could be used to collect all compiled modules and call the linker. In a sense, such global linking contradicts the modular extension-activation nature that Sugar* promotes. However, it would be possible to include some sort of *linker module* in a base language with the mere purpose of connecting otherwise unrelated modules. The method compile could implement a special handling for linker modules that initiates global compilation or linking for all required modules. This way we would circumvent the global nature of global compilation and would not require a closed world assumption. We will investigate this as part of our future work. For language extensions, we do not support separate compilation, since this would require an interface for the extensions against which clients of the extension can be compiled. Our extensions do not possess an interface useful for that purpose.

Interpreted (Ruby, Prolog): In interpreted languages, a module is not compiled but stored as source code for later execution. We support this feature by declaring method getBinaryFileExtension from interface IBaseLanguage as being optional and by allowing method compile to simply do nothing.

Summary. As our study of module-system features shows, Sugar* is able to support a large range of module-system features. This means that many base languages can be made extensible by instantiating Sugar*. However, for extension modules we are currently much more restrictive. To emphasize this difference, for some instantiations of Sugar* it would make sense to distinguish statements for base-language import from statements for extension import syntactically, for example using different keywords. In our future work, we want to investigate support base languages that use global or separate compilation as well as more flexible usage of extension modules.

7. Related work

Existing systems for syntactic abstraction typically fall into one of two categories: Either a system only supports a single base language, or it supports multiple base languages but is agnostic to the base language.

A prominent example of the first category are Scheme macros [5]. Scheme macros provide syntactic abstraction via compile-time functions that operator on abstract syntax trees. While the conceptual idea has been transferred to many base languages, the implementation of the Scheme macro system only supports Scheme as a base language. One of the reasons is that Scheme itself is used for implementing writing transformations of abstract syntax trees. In contrast, Sugar* employs metalanguages SDF, Stratego, and Spoofax editor specifications that are independent of the base language.

A prominent example of the second category is the C preprocessor (CPP) [25]. CPP supports the definition of lexical macros (#define) and compile-time conditionals (#ifdef). CPP supports multiple base languages; for example, it has been successfully applied in C, Java, and Haskell. However, CPP is agnostic to the base language. CPP operates on a stream of lexical tokens and is completely oblivious to the syntactic structure of the source code. In contrast, we require the definition of a parser for the base language and our transformations operate on abstract syntax trees of the base language.

Also most existing language workbenches fall into the second category. A language workbench [13, 16] is a tool that facilitates the definition of languages. While a language workbench allows the definition of multiple languages, the language workbench itself is agnostic to the actual workings of the defined language. Any language-specific functionality needs to be defined as part of the language definition itself, and the language workbench simply executes this functionality. In contrast, Sugar* has internal compilation logic that resolves modules of the base language, ensures sound recompilation schemes, and extracts and activates base-language extensions according to extension declarations and extension imports.

Polymorphic embedding [18] of domain-specific languages defines the notion of a language interface, but with a different meaning. In polymorphic embedding, the language interface declares the operators of the language and is parametric over the semantic domain that these operators result in. This allows the implementation of different semantics for a single language interface. In contrast, the language interface of Sugar* abstracts over the syntax and semantics of *different* base languages, where each base language is a separate instantiation of this interface.

Racket [14] allows the implementation of custom programming languages on top of Racket. Languages are defined as libraries [27]. They can later be used by using the #lang statement at the top of a source file. In Racket, the internal representation of each programming language is identical. The implementation of a language needs to provide a transformation that takes the source code as an input and outputs a transformed representation in Racket's internal language. Sugar* does not have an internal representation of programs written in a base language. Instead, the Sugar* compiler only interacts with the base language's module system and compiler via interfaces IBaseLanguage and IBaseProcessor.

Xbase [6] is a generic expression language for programming languages written using Xtext. Xbase abstracts over programming languages by offering a language-independent representation of expressions. We chose a different abstraction approach for Sugar* by offering a language-independent language-extension mechanism and compiler.

8. Conclusion

We presented Sugar*, a framework for syntactic language extensibility. Sugar* leverages the module system of the base language to encapsulate extensions as modules and to activate extensions via module import statements. To support language extensibility for many different base languages, we designed an abstract base-language representation that provides sufficient information for

tracking module dependency and for activating language extensions. Simultaneously, our abstract base-language representation is highly versatile and supports base languages with many different module-system features. To the best of our knowledge, Sugar* is the first system that supports syntactic language extensibility for a wide range of languages.

Acknowledgments

We are grateful to Florian Lorenzen for the development of SugarFomega and Bob Davison for the development of SugarJS. We thank the anonymous reviewers for their helpful remarks.

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] E. Burmako. Scala macros: Let our powers combine! In *Scala Workshop*, 2013. to appear.

[3] M. de Jonge. A pretty-printer for every occasion. In *Proceedings of Symposium on Constructing Software Engineering Tools (CoSET)*, pages 68–77, 2000.

[4] D. de Rauglaudre. Camlp4 reference manual. `http://caml.inria.fr/pub/docs/manual-camlp4/index.html`, accessed Mar. 26 2013, 2003.

[5] R. K. Dybvig, R. Hieb, and C. Bruggeman. Syntactic abstraction in scheme. *Lisp and Symbolic Computation*, 5(4):295–326, 1992.

[6] S. Efftinge, M. Eysholdt, J. Köhnlein, S. Zarnekow, R. von Massow, W. Hasselbring, and M. Hanus. Xbase: implementing domain-specific languages for java. In *Proceedings of Conference on Generative Programming and Component Engineering (GPCE)*, pages 112–121. ACM, 2012.

[7] S. Erdweg. *Extensible Languages for Flexible and Principled Domain Abstraction*. PhD thesis, Philipps-Universiät Marburg, 2013.

[8] 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.

[9] 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.

[10] 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.

[11] S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. Layout-sensitive generalized parsing. In *Proceedings of Conference on Software Language Engineering (SLE)*, volume 7745 of *LNCS*, pages 244–263. Springer, 2012.

[12] S. Erdweg, F. Rieger, T. Rendel, and K. Ostermann. Layout-sensitive language extensibility with SugarHaskell. In *Proceedings of Haskell Symposium*, pages 149–160. ACM, 2012.

[13] S. Erdweg, T. van der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. Vergu, E. Visser, K. van der Vlist, G. Wachsmuth, and J. van der Woning. The state of the art in language workbenches. In *SLE*, 2013. to appear.

[14] M. Flatt. Creating languages in Racket. *Communication of the ACM*, 55(1):48–56, 2012.

[15] M. Flatt, R. Culpepper, D. Darais, and R. B. Findler. Macros that work together—Compile-time bindings, partial expansion, and definition contexts. *Functional Programming*, 22(2):181–216, 2012.

[16] M. Fowler. Language workbenches: The killer-app for domain specific languages? Available at `http://martinfowler.com/articles/languageWorkbench.html`, 2005.

[17] O. Gruber, B. J. Hargrave, J. McAffer, P. Rapicault, and T. Watson. The Eclipse 3.0 platform: Adopting OSGi technology. *IBM Systems Journal*, 44(2):289–300, 2005.

[18] 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.

[19] 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.

[20] F. Lorenzen and S. Erdweg. Modular and automated type-soundness verification for language extensions. In *Proceedings of International Conference on Functional Programming (ICFP)*, 2013. to appear.

[21] OSGi Alliance. Osgi core release 5, 2012.

[22] F. Rieger. A language-independent framework for syntactic extensibility. Bachelor's Thesis, University of Marburg, June 2012.

[23] T. Sheard and S. Peyton Jones. Template meta-programming for Haskell. In *Proceedings of Haskell Workshop*, pages 1–16. ACM, 2002.

[24] K. Skalski, M. Moskal, and P. Olszta. Meta-programming in nemerle. `http://nemerle.org/metaprogramming.pdf`, accessed Oct. 01 2012., 2004.

[25] R. Stallman and Z. Weinberg. The C Preprocessor. Available at `http://gcc.gnu.org/onlinedocs/cpp/`, accessed Nov. 08, 2012., 1987.

[26] 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.

[27] 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)*, pages 132–141. ACM, 2011.

[28] M. van den Brand and E. Visser. Generation of formatters for context-free languages. *Transactions on Software Engineering Methodology (TOSEM)*, 5(1):1–41, 1996.

[29] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, 1997.

[30] E. Visser, Z.-E.-A. Benaissa, and A. P. Tolmach. Building program optimizers with rewriting strategies. In *Proceedings of International Conference on Functional Programming (ICFP)*, pages 13–26. ACM, 1998.

[31] D. Weise and R. F. Crew. Programmable syntax macros. In *Proceedings of Conference on Programming Language Design and Implementation (PLDI)*, pages 156–165. ACM, 1993.