SugarJ: Library-based Syntactic Language Extensibility

Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann

University of Marburg, Germany

Abstract

Existing approaches to extend a programming language with syntactic sugar often leave a bitter taste, because they cannot be used with the same ease as the main extension mechanism of the programming language—libraries. *Sugar libraries* are a novel approach for syntactically extending a programming language within the language. A sugar library is like an ordinary library, but can, in addition, export syntactic sugar for using the library. Sugar libraries maintain the composability and scoping properties of ordinary libraries and are hence particularly well-suited for embedding a multitude of domain-specific languages into a host language. They also inherit the self-applicability of libraries, which means that the syntax extension mechanism can be applied in the definition of sugar libraries themselves.

To demonstrate the expressiveness and applicability of sugar libraries, we have developed *SugarJ*, a language on top of Java, SDF and Stratego that supports syntactic extensibility. SugarJ employs a novel incremental parsing technique that allows changing the syntax within a source file. We demonstrate SugarJ by five language extensions, including embeddings of XML and closures in Java, all available as sugar libraries. We illustrate the utility of self-applicability by embedding XML Schema, a metalanguage to define XML languages.

1. Introduction

Bridging the gap between domain concepts and the implementation of these concepts in a programming language is one of the "holy grails" of programming. Domain-specific languages (DSLs), such as regular expressions for the domain of textual querying or Java Server Pages for the domain of dynamic web pages have often been proposed to address this problem [28]. To use DSLs in large software systems that touch multiple domains, developers have to be able to compose multiple domain-specific languages and embed them into a common host language [21]. In this context, we consider the long-standing problem of domain-specific *syntax* [4, 5, 8, 26, 34, 48].

```
import pair.Sugar;

public class Test {
  private (String, Integer) p = ("12", 34);
}
```

Figure 1. Using the pair sugar library.

Our novel contribution in this domain is the notion of *sugar libraries*, a technique to syntactically extend a programming language in the form of libraries. In addition to the semantic artifacts conventionally exported by a library, such as classes and methods, sugar libraries export also syntactic sugar that provides a user-defined syntax for using the semantic artifacts exported by the library. Each piece of syntactic sugar defines some extended syntax and a transformation—called *desugaring*—of the extended syntax into the syntax of the host language. Sugar libraries enjoy the same flexibility as conventional libraries: (i) They can be used where needed by importing the syntactic sugar as exemplified in Fig. 1. (ii) The syntax of multiple composable DSLs can be composed by importing all corresponding sugar libraries. Their composition may form a new higher-level DSL that can again be packaged as a sugar library. (iii) Sugar libraries are self-applicable: They can import other sugar libraries, which means that the syntax for specifying syntactic sugar can be extended as well.

In other words, SugarJ treats language extensions in a unified and regular fashion at all metalevels. Here, we apply a conceptual understanding of "metalevel", which distinguishes the definition of a language from its usage: a language definition is on a higher metalevel than the programs written in that language. In this sense, sugar libraries (defining language extensions) are on a higher metalevel than the programs that use the sugar library, and the import of a sugar library acts across metalevels.

Sugar libraries are not limited to DSL embeddings; they can be used for arbitrary extensions to the surface syntax of a host language (for instance, an alternative syntax for method

```
package pair;
public class Pair<A,B> { ... }
```

(a) We implement the semantics of pairs as a generic Java class.

```
package pair;
import org.sugarj.languages.Java;
import concretesyntax.ConcreteJava;
public sugar Sugar {
 context-free syntax
  "(" JavaType "," JavaType ")" -> JavaType{cons("PType")}
  "(" JavaExpr "," JavaExpr ")" -> JavaExpr{cons("PExpr")}

 desugarings
  desugar-pair-type
  desugar-pair-expr

 rules
  desugar-pair-type :
   PType(t1, t2) -> |[ pair.Pair<~t1, ~t2> ]|

  desugar-pair-expr :
   PExpr(e1, e2) -> |[ pair.Pair.create(~e1, ~e2) ]|
}
```

(b) Extending the Java syntax and specifying desugaring rules.

Figure 2. Sugar libraries comprise syntax and semantics.

calls). However, due to their composability and their alignment with the import and export mechanism of libraries, they qualify especially for embedding DSLs.

To explore sugar libraries, we have designed and implemented sugar libraries in *SugarJ*. SugarJ is a programming language based on Java that supports sugar libraries by building on the grammar formalism SDF [18] and the transformation system Stratego [43]. As an example of SugarJ's syntactic extensibility, in Fig. 1 we import a sugar library for pairs that enables us to use pair expressions and types with pair-specific syntax. The corresponding sugar library is illustrated in Fig. 2 and consists of two components: a generic class Pair<A,B>, which implements pair expressions and types semantically, and syntactic sugar pair.Sugar that provides a pair-specific syntactic interface for the library. The pair.Sugar declaration extends the Java syntax with syntax for pair types and expressions and stipulates how pair syntax is desugared to Java. In Fig. 1, for example, desugaring transforms the pair type (String, Integer) into the Java type Pair<String, Integer> and the pair expression ("12", 34) into a static method call pair.Pair.create("12", 34). Since SugarJ supports arbitrary compile-time computation, sugar libraries can implement even intricate source transformations or perform domain-specific compile-time checking.

After briefly reviewing the syntactic extensibility of existing DSL embedding approaches, we make the following contributions:

- We introduce the novel concept of sugar libraries, a *library-centric* approach for syntactic extensibility of host languages (Section 3). Sugar libraries enable the uniform embedding of DSLs at syntactic and semantic level, and retain (with some limitations discussed in Section 6.1) the composability properties of conventional libraries.
- Sugar libraries combine the benefits of existing approaches: Sugar libraries support flexible domain-specific syntax (with arbitrary context-free grammars and compile-time checks) *and* can be imported across metalevels to activate language extensions in user programs *and* act on all metalevels uniformly to enable syntactic extensions in metaprograms (self-applicability).
- The *easy* mechanism to use syntactic extensions by simple import statements and the language support to develop new syntactic extension, even for small language extensions, encourages development in a language-oriented [47] fashion.
- We present our implementation of SugarJ[1] on top of existing languages, namely Java, SDF and Stratego, and explain the mechanics of compiling our syntactically extensible programming language (Section 4).
- Technically, we present an innovative incremental way of parsing files, in which different regions of a file use different syntactic extensions.
- We demonstrate the expressiveness and applicability of SugarJ on the basis of five case studies – pairs, closures, XML, concrete syntax in transformations, and XML Schema – the latter being an advanced example of self-applicability, since each XML Schema defines a new XML language (Section 5).

2. Syntactic embedding of DSLs

Many approaches for embedding DSLs into a host language mostly focus on the integration of domain concepts at semantic level (e.g., [19, 20, 32]), but neglect the need for expressing domain concepts using domain syntax. To set the context for sugar libraries, we survey the syntactic amenability of existing DSL embedding approaches, while a more thorough treatment of related work appears in Section 7.

String encoding. The simplest form of representing a DSL program in a host language is as unprocessed source code, encoded as a host language string. Since most characters may occur in strings freely, such encoding is syntactically flexible. Consider, for instance, the following Java program which writes an XML document to some output stream out.

```
String title = "Sweetness and Power";
out.write("<book title=\"" + title + "\">\n");
out.write("  <author name=\"Sidney W. Mintz\" />\n");
```

[1] The source code of our SugarJ implementation and all case studies is available at `http://sugarj.org`.

```
out.write("</book>");
```

The string encoding allows writing XML code with tags and attributes naturally. Nevertheless, in XML documents nested quotes and special whitespace symbols such as newline have to be escaped, leading to less legible code. Moreover, the syntax of string encoded DSL programs is not statically checked but parsed at runtime; hence, syntactic errors are not detected during compilation and can occur after deploying the software. Furthermore, string encoded programs have no syntactic model and, thus, can only be composed on a lexical level by concatenating strings. This form of composition resembles lexical macro expansion in a way that is not amenable to parsing [14] and opens the door to security problems such as SQL injection or cross-site scripting attacks [7].

Library embedding. To avoid lexical composition and syntax errors at runtime, we can alternatively embed a DSL as a library, that is, a reusable collection of functionality accessible through an API. In Hudak's pure-embedding approach [21], for instance, one builds a library whose functions implement DSL concepts and are used to describe DSL programs. For example, we can embed XML purely as follows:

```
String title = "Sweetness and Power";
Element book =
  element("book",
    attributes(attribute("title", title)),
    elements(
      element("author",
        attributes(attribute("name", "Sidney W. Mintz")),
        elements())));
```

The syntax of the DSL can be encoded in the type system of the host language, so that, in a statically typed host language, the DSL program is syntax checked at compile-time. In our example, such checks can prevent confusion of attributes and elements, for instance. Even in an untyped language, purely embedded XML documents are properly nested by design, that is, it is not possible to describe ill-formed documents such as <a>.

An apparent drawback of purely embedded DSLs is the syntactic inflexibility of the approach: Programmers must adopt the syntax of function calls in the host language to describe DSL programs. Consequently, when solving a problem in terms of a certain domain, the programmer needs to "translate" the proposed solution into the host language's syntax manually. Some host languages partially address this problem by overloading of, for instance, built-in or user-defined infix operators, integer or string literals, or even function calls. Even in these languages, however, a DSL implementer can only extend the host language's syntax in a limited, preplanned way. For example, while Scala supports quite flexible syntax for method calls, the syntax for class declarations is fixed.

To circumvent the need for manual translation of domain concepts, researchers have proposed the use of syntactically extensible host languages that support the syntactic embedding of DSLs [3, 5, 40, 48]. In particular, languages with macro facilities (or similar metaprogramming facilities) can be used to develop library-based syntactic embeddings of DSLs [25]. Unfortunately, most macro languages only support user-defined syntax for macro arguments [5]. This obstructive requirement for explicit macro invocations prevents the usage of macro systems to syntactically embed DSLs into a host language freely [8].

Independent of their syntactic inflexibility, one essential advantage of library embeddings is the composability of DSLs. By importing multiple libraries, a programmer can easily compose those libraries to build a new one. Since embedded DSLs are implemented as libraries of the host language, library composition entails the composition of DSL implementations. Therefore, library embedding supports modular definitions of DSLs on top of previously existing ones [20]. These benefits of library embedding are the starting point and main motivation for our sugar-library approach.

Language extension. To support statically-checked and domain-specific syntax, one possibility is to extend the host language such that it comprises the DSL. In this approach, syntactic and semantic language extensions are incorporated into the host language by directly modifying its implementation or using an extensible compiler. Usually, language extensions are not restricted in the syntax they introduce; thus, DSL implementors can integrate arbitrary DSL syntax and semantics into the host language. For example, Scala provides built-in support for XML documents:

```
val title = "Sweetness and Power"
val book =
  <book title="{title}">
    <author name="Sidney W. Mintz" />
  </book>
```

Scala's support for XML syntax has been directly integrated into the Scala compiler, which translates XML syntax trees into calls to the scala.xml library [31]. Since the Scala compiler parses embedded XML documents at compile-time, runtime syntax errors cannot occur and ill-formed documents cannot be generated. However, in contrast to purely embedded DSLs, users of an XML-extended host language can write programs using XML syntax more naturally, compared to nested library calls.

In general, modifying a (non-extensible) compiler to incorporate a DSL into the host language is impracticable and makes it hard to develop compose independent DSLs. More generic approaches for extending a language support the modular definition and integration of DSLs and are not specific to the used host language. In these approaches, which include extensible compilers [13, 30] and program transformation systems [8, 42], the used language extensions are de-

termined by compiler configurations or by generating and selecting the right compiler variant. This becomes impractical if multiple DSLs are used, because compiler variants or configurations have to be generated for each combination of DSLs, and a significant part of the program's semantics and dependency structure is moved from the program sources to build or configuration scripts.

Summary. String embedding is syntactically very flexible but lacks static safety and composability. Library embeddings excel in composability but lack syntactic flexibility. Language extensions are powerful but hard to implement and compose and introduce an undesirable stratification into base code and meta-level code. Obviously, it would be beneficial to combine their respective strengths.

3. SugarJ: Sugar libraries for Java

We propose to organize syntactic language extensions into sugar libraries that encapsulate specifications of syntax extensions and their desugaring into a host language. To use a sugar library, a developer simply imports the library and may use the new syntax constructs in subsequent segments of the same file. Programmers and metaprogrammers can uniformly import sugar libraries to implement applications or other sugar libraries.

To demonstrate the concept, we have designed and implemented SugarJ, a variant of Java with support for sugar libraries. The design and implementation of SugarJ is based on three existing languages: Java is used as host language for application code, the syntax definition formalism (SDF) [18] is used to describe concrete syntax, and the Stratego transformation language [43] is used to desugar extension-specific code into SugarJ code. In particular, extension-specific code can not only desugar into Java, but also into SDF or Stratego fragments which define another extension. This reuse of existing technology enables us to implement a fully-featured and highly expressive prototype of SugarJ, while still concentrating on the novel aspects of its language design and implementation.

We introduce sugar libraries by walking through an example. We extend Java with closures by introducing syntactic sugar and corresponding translations to desugar the introduced closure syntax into plain Java code. (Closures, or lambda expressions or anonymous functions, are an often requested feature for Java and plans exist to integrate closures into Java 8, which is expected for late 2012.)

3.1 Using a sugar library

To use a sugar library, the only thing a programmer has to do is to import the library with an ordinary `import` statement. In a file that imports a sugar library, the programmer may use syntax introduced by the library anywhere after the import. All syntax constructs from the library are desugared into plain Java code (more precisely into SugarJ code, because desugarings can produce new syntax extensions) automatically at compile-time.

```
package javaclosure;
public interface Closure<Result, Argument> {
  public Result invoke(Argument argument);
}
```

(a) An interface for function objects.

```
final int factor = ...;
Closure<Integer, Integer> closure =
  new Closure<Integer, Integer>() {
    public Integer invoke(Integer x) {
      return x * factor;
    }
  };
List<Integer> scaled = original.map(closure);
```

(b) Creating a closure.

Figure 3. Closures can be implemented as function objects, but Java does not offer convenient syntax for closure expressions.

```
import javaclosure.Syntax;
import javaclosure.Desugar;
```

(a) Importing the sugar library for closures.

```
final int factor = ...;
#Integer(Integer) closure =
  #Integer(Integer x) {return x * factor;};
List<Integer> scaled = original.map(closure);
```

(b) Creating a closure.

Figure 4. Specialized syntax for closures with SugarJ.

Our closure example illustrates the benefits of sugar libraries for programmers and how easy such libraries are to use. In plain Java code, a programmer would typically implement closures as anonymous inner classes as illustrated in Fig. 3(b). However, the syntax is rather verbose, especially for the frequent use case of an anonymous inner class with exactly one method. With SugarJ, we simply import a sugar library that introduces a more concise notation for closures, following roughly the proposal of Gafter and von der Ahé [17] (one of several syntax suggestions). With this library, we can rewrite our example as illustrated in Fig. 4: Instead of verbose plain Java code, we write `#R(T)` for denoting closure types `Closure<R, T>` and `#R(T t) { stmts...; return exp; }` for defining a closure. Both code fragments are equivalent. SugarJ automatically desugars the concise version into plain Java code at compile-time.

3.2 Writing a sugar library

To write a sugar library, one has to define how to extend the language and how to desugar the extension. Hence, also a sugar library conceptually consists of two parts: An extension of the host language's grammar with new syntax rules, and a desugaring of the new language constructs into the original language.

In SugarJ, programmers define both parts through top-level sugar declarations of the form **public sugar** Name { ... }, which contain SDF and Stratego code organized into sections. All valid SDF and Stratego sections can be used in a sugar declaration, but we concentrate on the features most essential in writing sugar libraries: Syntax rules and desugaring rules.

In an SDF section **context−free syntax**, a library developer can extend the host language's grammar with new syntax rules. We illustrate the corresponding extensions for our closure example in Fig. 5(a). A syntax rule specifies the non-terminal to be extended (to the right of the arrow $->$), a pattern for the newly introduced concrete syntax (to the left of the arrow $->$), and a name for the syntax tree node created by this production (in the cons annotation). In this way, sugar libraries can introduce new syntax by extending SugarJ non-terminals or non-terminals introduced by other sugar libraries.

Analogously, in a Stratego section **rules**, a library developer can define program transformations, called desugaring rules. We illustrate the rules for closures in Fig. 5(b). A desugaring rule consists of a name (before the colon), a matching pattern (to the left of the arrow $->$) and a generation template (to the right of the arrow $->$). Both pattern and template are specified using concrete syntax in brackets |[...]|, where meta-variables are written with an initial tilde ∼. A desugaring rule denotes a program transformation from the extended to the original language, or to the original language with some other extension.

Desugaring rules are specified using concrete syntax, so that a programmer does not need to read or write any abstract syntax trees. In our example, the rule desugar−closure−type in Fig. 5(b) matches on closure types using the # ... (...) concrete syntax just introduced in Fig. 5(a). For technical reasons, however, a syntax rule is only activated *after* the **sugar** declaration where it is defined.[2] Therefore one typically splits a sugar library into two parts, introducing syntax rules and desugaring rules separately, so that the syntax rules for closures are in scope when we define the desugaring rules for closures.

In a final section **desugarings** of the sugar library, the library developer declares the entry-points for desugaring. After parsing, the SugarJ compiler exhaustively applies these desugaring rules in a bottom-up fashion, starting at the syntax tree's leaves and progressing towards its root. Compilation fails if an input program cannot be unambiguously parsed with the combination of all syntax rules in scope, if any of the triggered desugaring rules signals an error, or if the desugared program still contains fragments of user extensions.

```
package javaclosure;
import org.sugarj.languages.Java;
import concretesyntax.ConcreteJava;

public sugar Syntax {
  context−free syntax
    "#" JavaType "(" JavaType ")"
      −> JavaType {cons("ClosureType")}

    "#" JavaType "(" JavaFormalParam ")" JavaBlock
      −> JavaExpr {cons("ClosureExpr")}
}
```

(a) Extending the Java grammar

```
public sugar Desugar {
  rules
    desugar−closure−type :
      |[ #∼result(∼argument) ]|
      −> |[ javaclosure.Closure
              <? extends ∼result, ? super ∼argument> ]|

    desugar−closure−expr : ... −> ...

  desugarings
    desugar−closure−type
    desugar−closure−expr
}
```

(b) Desugaring closures.

Figure 5. Introducing syntactic sugar for closures. The sugar library is split over two **sugar** declarations so that the syntax rules from (a) are in scope in (b).

3.3 Composing sugar libraries

Sugar libraries are composed by importing more than one sugar library into the same file. For example, in Fig. 6, we import the sugar library for closures together with a sugar library for pairs to implement partial application of a function that expects a pair as input. Even though javaclosure.Syntax and javaclosure.Desugar belong together, SugarJ does not allow to import them with a single statement, because the SugarJ import statement is modeled after the Java import statement, which does not support such compound modules, either.[3] The scope of each sugar library is annotated in the

[2] Our implementation supports syntax changes only between top-level declarations, but not in the middle of, for example, a **sugar** declaration. See Sections 3.3 and 4 for details.

[3] Java supports wildcard imports like **import** javaclosure.∗, but their semantics is ill-suited for our purpose: A wildcard import only affects unqualified class names, but the name of a sugar library never occurs in a source

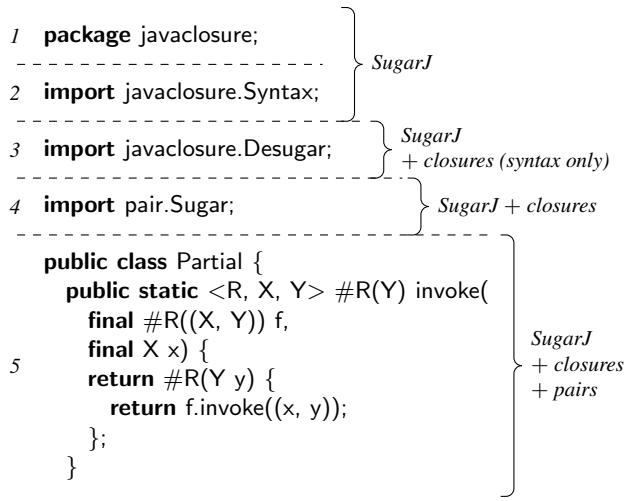

Figure 6. Composing two sugar libraries by importing both.

figure. The syntax for closures and the syntax for pairs can be freely mixed in the class declaration, where both sugar libraries are in scope. Composing two sugar libraries is not always possible entirely without conflicts or ambiguities if the syntactic extensions overlap. Our experience, however, shows that in most practical cases libraries can be freely composed, or conflicts can be easily detected and fixed, see our discussion in Section 6.1.

4. SugarJ: Technical realization

A compiler for SugarJ parses and desugars a SugarJ source file and produces a Java file together with grammar and desugaring rules as output. Subsequently, we can compile the Java file into byte code, whereas the grammar and desugaring rules are stored separately as a form of library interface for further imports from other SugarJ files. In this section, we assume that desugaring rules are program transformations between syntax trees. Later, in Section 5.1, we show how an ordinary sugar library can extend SugarJ to support desugarings rules in terms of concrete syntax.

4.1 The scope of sugar libraries

To parse and desugar a SugarJ source file, the compiler keeps track of which grammar and desugaring rules apply to which parts of the source file. Through importing or defining a sugar library, the grammar and desugaring rules may change within a single source file. Moreover, definitions and import statements of sugar libraries may in turn be written using syntactic sugar and therefore have to be desugared before continuing with parsing.

file. Instead, the SugarJ compiler needs to immediately import the sugar library to parse the next top-level declaration with an updated grammar. It would obviously be desirable to have a grouping construct with which multiple associated syntax definitions can be imported at once, but for this work we decided to stay close to the Java import model and leave such an extension for future work.

In SugarJ, imports and declarations of sugar libraries can only occur at the top-most level of files, but not nested inside other declarations. The scope of grammar and desugaring rules therefore always aligns with the top-level structure of a file. For example, in Fig. 6, the grammar and desugaring rules change between the the second and the third top-level entry for the first time, hence the third top-level entry is parsed and desugared in a different context. Subsequently, it changes again after the third and after the fourth top-level entry, which influences parsing and desugaring of the remaining file. This alignment allows the SugarJ compiler to interleave parsing and desugaring at the granularity of top-level entries.

4.2 Incremental processing of SugarJ files

Our SugarJ compiler parses and desugars a SugarJ source file one top-level entry at a time, keeping track of changes to the grammar and desugaring rules which will affect the processing of subsequent top-level entries. A top-level entry in SugarJ is either a package declaration, an import statement, a Java type declaration, a declaration of syntactic sugar or a user-defined top-level entry, introduced with a sugar library. As illustrated in Fig. 7, the compiler processes each top-level declaration in four steps: parsing, desugaring, splitting and adaption.

Parsing. Each top-level entry is parsed using the *current grammar*, that is, the grammar which reflects all sugar libraries currently in scope. For the first top-level entry, the current grammar is the initial SugarJ grammar, which comprises Java, SDF and Stratego syntax definitions. For subsequent top-level entries, the current grammar may differ due to declared or imported syntactic sugar. The result of parsing is a heterogeneous abstract syntax tree, which can contain both predefined SugarJ nodes and user-defined nodes.

Desugaring. Next, the compiler desugars user-defined extension nodes of each top-level entry into predefined SugarJ nodes using the *current desugaring*. For each top-level entry, the current desugaring consists of the desugaring rules currently in scope, that is, the desugaring rules from the previously declared or imported sugar libraries. Desugarings are transformations of the abstract syntax tree, which the compiler applies in a bottom-up order to all abstract-syntax-tree nodes until a fixed point is reached. The result of this desugaring step is a homogeneous abstract syntax tree, which contains only nodes declared in the initial SugarJ grammar (if some user-specific syntax was not desugared, the compiler issues an error message). Thus, this tree represents one of the predefined top-level entries in SugarJ and is therefore composed only of nodes describing Java code, grammar rules or desugarings. These constituents can now be splitted to yield three separate artifacts.

Splitting. SugarJ is based on three languages, namely Java, SDF and Stratego for denoting general computation, syntax

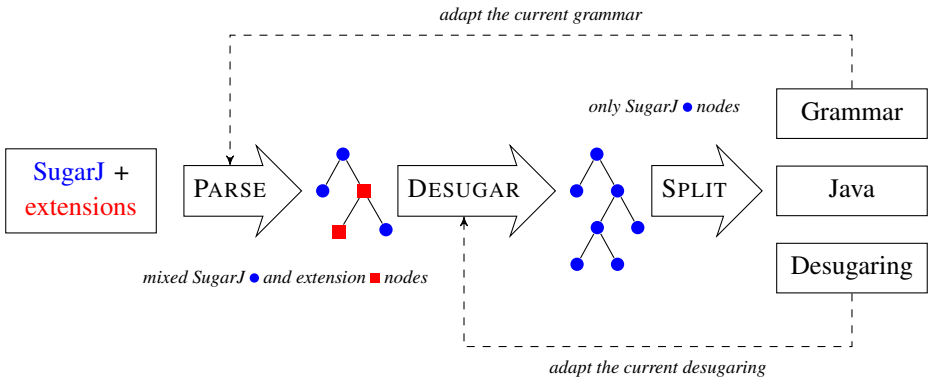

Figure 7. Processing of a SugarJ top-level form.

definitions and compile-time transformations, respectively. We split each top-level SugarJ declaration into fragments of these languages and reuse their respective implementations.

For a SugarJ package declaration, we write corresponding package declarations into all output files to synchronize the corresponding module systems (more on that later). Import declarations are propagated to the Java output and may affect the module system of grammars and desugarings as well. In contrast, a class declaration is written only into the Java output and a sugar declaration affects the grammar specification and desugaring file.

After the last top-level declaration, the Java file contains pure Java code and the grammar specification and desugaring rules are written in a form that can be imported by other SugarJ files. In case any produced artifact does not compile, the SugarJ compiler issues a corresponding error message.

Adaption. As introduced above, sugar declarations and imports can affect the parsing and desugaring of all subsequent code in the same file. Therefore, after each top-level entry, we reflect possible syntactic extensions by adapting the *current* grammar and the *current* desugaring.

If a top-level declaration in the desugared abstract syntax tree is a new sugar declaration, we (a) *compose* the current grammar with the grammar of the new declaration and (b) *compose* the current desugaring rules with the desugaring rules of the new declaration. If the top-level declaration is an import declaration, we load the corresponding sugar library from the class path and compose its extensions of grammar and desugaring with the current grammar and desugaring. On pure Java declarations, we do not need to update the current grammar or desugaring.

When composed, productions of two grammars (e.g., from the initial SugarJ grammar and from a grammar in a sugar library) can interact through the use of shared nonterminal names. Hence, a sugar library can add productions to nonterminals that were originally defined either in the initial grammar or in some other sugar library. In that way, nonterminals defined in the initial grammar are initial extensions points for grammar rules defined in sugar libraries. Similarly, when composed, two sets of desugaring rules can interact through the use of shared names and by producing abstract-syntax-tree nodes that are subsequently desugared by rules from the other set.

Adaptation and composition of grammar and desugarings may take place after every top-level declaration and affect the processing of all subsequent top-level declarations.

4.3 The implementation of grammars and desugaring

As mentioned earlier, SugarJ uses the syntax definition formalism SDF [18] to represent and implement grammars and the transformation language Stratego [43] to represent and implement desugarings.

The initial grammar (with regard to the process described in Section 4.2) is a standard Java 1.5 grammar augmented by top-level sugar declarations. To enable incremental parsing with different grammars, we have adapted the Java grammar by a non-terminal which parses a single top-level entry together with the rest of the file as a single string.

Before using SDF grammars and Stratego transformations, they have to be compiled. Our implementation caches the results of compiling composed SDF and Stratego programs to speed up the usual case of using the same combination of sugar libraries multiple times, either processing different files using the same set of sugar libraries, or reprocessing the same file after changes which do not affect the imports. In such a case, our compiler takes only a few seconds to compile a SugarJ file. In contrast, when changing the language of a SugarJ file, all syntax rules and desugaring rules in scope are recompiled, thus compilation takes considerably longer (but typically well under a minute). Separate compilation [10] would help to speed up compilation, but SDF and Stratego traditionally focus on the flexible combination of modules, not on compiling them separately.

5. Case studies

Our primary goal in designing SugarJ is to support the integration and composition of DSLs at semantic and syntactic level. To this end, we provide SugarJ with an extensible sur-

face syntax that sugar libraries can freely extend to embed arbitrary domain syntax.

We have embedded a number of language extensions and DSLs into SugarJ, including syntax for pair expression and pair types (Section 1), closures for Java (Section 3) and regular expressions. All of these case studies are implemented in similar style: define an extended syntax and its desugaring into an existing Java implementation for the domain. In this fashion, we could have easily embedded many more DSLs such as Java Server Pages or SQL. Many such case studies have been performed for MetaBorg [8]; since we use the same underlying languages for describing grammars and desugarings, namely SDF and Stratego, these embeddings could easily be encoded as sugar libraries by lifting the implementations into SugarJ's syntax and module system. In contrast to the case studies in MetaBorg, the resulting SugarJ libraries can be activated across metalevels and composed by issuing import instructions and need neither complicated compiler configurations nor explicit compound modules. Due to the simplicity of activating sugar libraries, they are not only well-suited for large-scale embeddings of DSLs but also for using several small language extensions such as pairs and closures.

Since the embedding of further "ordinary" DSLs is not likely to yield more insight, we focus our attention on more sophisticated scenarios that demonstrate the flexibility of sugar libraries compared to other technologies. In the pair and closure case studies, we already used a sugar library that provides concrete syntax for implementing program transformations. We will explain this sugar library in the following subsection. Subsequently, we focus on the composability features of SugarJ by discussing an embedding of XML syntax into SugarJ, which reuses existing sugar libraries in non-trivial ways. We close the present section by elaborating on SugarJ's support for implementing meta-DSLs, that is, special-purpose languages for implementing DSLs. Specifically, we embed XML Schema into SugarJ to describe languages of valid XML documents for which validation is a compile-time check.

5.1 Concrete syntax in transformations

As described in Section 4, the SugarJ compiler parses a SugarJ top-level declaration into an abstract syntax tree before applying any desugaring rules. Desugaring rules are therefore expressed internally as transformation between abstract syntax trees, even when they are *specified* in terms of concrete syntax, as described in Section 3.2. Concrete syntax in transformations, however, significantly increases the usability of SugarJ: A sugar library developer, who wants to extend the visible surface syntax, should not need to reason about the underlying invisible abstract structure.

To support concrete syntax in transformations, we could have changed the SugarJ compiler, leading to a monolithic and not very flexible design. The self-applicability of SugarJ, however, allows a more flexible and modular solution: We implement concrete syntax in transformations as a sugar library concretesyntax.ConcreteJava that extends the syntax for the specification of sugar libraries itself. We have imported this sugar library in the sugar libraries for pairs and closures above.

For example, the desugaring rules for pair expressions can conveniently be written as a transformation between snippets of concrete syntax as follows:

```
desugar−pair :
  |[ (∼expr:e1, ∼expr:e2) ]| −>
  |[ pair.Pair.create(∼e1, ∼e2) ]|
```

This rule is desugared into a transformation between abstract syntax trees as follows:

```
desugar−pair:
  PExpr(e1, e2) −>
  Invoke(
    Method(MethodName(
      AmbName(AmbName(Id("pair")), Id("Pair")),
      Id("create")))),
    [e1, e2])
```

Visser proposed the use of concrete syntax in the implementation of syntax tree transformation [44] for MetaBorg. Technically, a transformation that uses concrete syntax expands to a transformation with abstract syntax by parsing the concrete syntax fragments and injecting the resulting abstract syntax tree. Thus, the left-hand and right-hand sides of the former desugar−pair transformation expand to the ones of the latter transformation. This technique is language-independent and has been implemented generically [44] such that the concrete syntax of any language can be injected into Stratego by extending Stratego's grammar accordingly. For example, to enable concrete syntax for Java expressions in transformations, the following production can be introduced to specify that quoted Java code is written in brackets |[... |] and anti-quoted Stratego code is preceded by a tilde ∼.

```
"|[" JavaExpr "]|" −> StrategoTerm {cons("ToMetaExpr")}
"∼" StrategoTerm −> JavaExpr {cons("FromMetaExpr")}
```

In SugarJ, the sugar library for concrete syntax in transformations, whenever it is in scope, automatically desugars concrete syntax into abstract syntax as described above. In contrast, in MetaBorg, the desugaring of concrete syntax is a preprocessing step which the programmer needs to enable manually by accompanying the Stratego source file with an equally named "`*.meta`" file pointing to the SDF module used for desugaring [44]. The reason for this obstructive mechanism is that support for concrete syntax is syntactic sugar at metalevel. Due to the homogeneous integration of metalanguages in SugarJ, however, SugarJ is host and metalanguage at the same time. Therefore, language extensions of SugarJ can be developed as sugar libraries in SugarJ itself.

```
import xml.XmlJavaSyntax;
import xml.AsSax;
```

(a) Importing the XML syntax and desugaring.

```
public void genXML(ContentHandler ch) {
  String title = "Sweetness and Power";
  ch.<book title="{title}">
       <author name="Sidney W. Mintz" />
     </book>;
}
```

(b) Generating an XML document using XML syntax. The anti-quote operator {...} allows SugarJ code to occur inside XML documents.

Figure 8. XML documents are statically syntax checked and desugared to calls of SAX.

The alignment of metalanguage and host language in SugarJ implicates that a programmer can develop and apply language extensions within a single language, and never has to specify any configuration external to the language such as a build script or a "`*.meta`" file. This has a fundamental consequence, namely it enables programmers to conduct modular reasoning: Every fact about a given SugarJ program is derivable from its source code and the modules it references; it is not necessary to take build scripts or configuration files or, in fact, any code into account that is not referenced within the source file. This becomes particularly important when the number of available DSLs grows, as, for instance, in the implementation of the XML sugar library.

5.2 XML documents

The embedding of XML [46] syntax, as discussed in Section 2, is a good show-case for syntactic extension: Many existing APIs for XML suffer from a significant syntactical overhead over direct use of XML tag notation, XML syntax does not follow the lexical structure of most host languages, and neither well-formedness nor validation of XML documents are context-free properties. The implementation of our sugar library for XML syntax furthermore serves as an example to discuss SugarJ's support for modularity.

Typically, XML is integrated into a host language through the provision of an API such as the Simple API for XML (SAX) or the Document Object Model. Following the MetaBorg XML embedding [8], our sugar library for XML syntax desugars XML syntax into an indirect encoding of documents through SAX calls. In Fig. 8, for example, an XML document is sent to a content handler `ch`. Compared to Scala's XML support (Section 2), sugar libraries provide similar syntactic flexibility without changing the host language's compiler.

The XML sugar library statically ensures that all generated XML documents are well-formed and, to this extent, supports the same static checks as the pure embedding approach shown in Section 2. However, the SAX API, due to its event-driven design, does, for instance, not statically detect illegal nesting as in `<a><b></a></b>` or mismatching start and end tags as in `<a></b>`. The XML sugar library arranges to check both properties, the former during parsing and the latter during a separate checking phase.

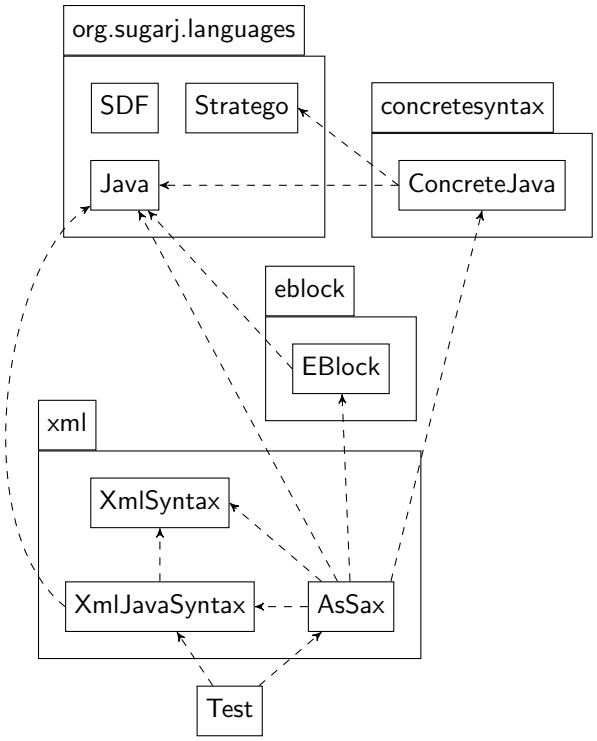

Figure 9. The structure of the XML case study: arrows depict dependencies between sugar libraries and are resolved through sugar library imports.

This introduces an interesting distinction of the kind of static checks we can perform in sugar libraries. On the one hand, context-free properties such as legal nesting of XML elements can be encoded into the syntax definition of a language extension, and the compiler checks them while parsing the source code. On the other hand, context-sensitive properties cannot be encoded into context-free syntax rules; instead, it is possible to encode the checking of context-sensitive properties as a program transformation that traverses the non-desugared syntax tree and generates error messages as needed. For example, the XML sugar library contains a compile-time check that verifies that all XML elements have equal start and end tags. Consequently, elements with mismatching tags are detected at compile-time and lead to a compiler error as expected.

When developing the XML sugar library, we heavily reused other sugar libraries at metalevel in non-trivial ways, including the library for concrete syntax from the previous subsection. The diagram in Fig. 9 depicts the structure and dependencies of the components involved in embed-

ding XML. The xml package contains three sugar declarations. XmlSyntax defines the abstract and concrete syntax of XML, which is embedded into the syntax of Java by XmlJavaSyntax. AsSax defines how to desugar an XML document into a sequence of SAX library calls. Since XML documents are integrated into Java at expression level, whereas the SAX library is accessed via statements, calls to SAX have to be lifted from expression level to statement level. To this end, we adopted the use of expression blocks EBlock from MetaBorg [8]. Finally, AsSax uses concrete syntax and expression blocks to generate Java code.

Evidently, composing and reusing language extensions is essential in the implementation of XML. Since in SugarJ the primary means of organizing language extensions and DSLs are libraries, programmers can import sugar libraries to build their DSL or language extension on top of existing ones. In the implementation of AsSax, for instance, we desugar XML trees into Java with expression blocks. The concrete syntax of expression blocks from EBlocks is directly available in desugaring rules, even though the support for concrete syntax in transformations was defined independently in ConcreteJava. This is possible because both sugar libraries extend the same Java non-terminals imported from Java. In general, however, and as for ordinary libraries, it might be necessary to write "glue code" to compose individual sugar libraries meaningfully. Regarding the limits of sugar library composition see also the discussion in Section 6.1.

The XML case study illustrates how sugar libraries can be composed to make joint use of distinct syntactic extensions. It is important to note that the embedding of XML is not the end of the line of extensibility, but is itself a sugar library that can be utilized in further extending the language. We will demonstrate this feature in the following case study, where we implement a type system for XML documents as a sugar library.

5.3 XML Schema

A meta-DSL is a DSL in which other DSLs can be defined. SugarJ supports the definition of meta-DSLs naturally since it enables syntactic extensions of the metalanguage and the host language. Sugar libraries can thus provide new "frontends" for building other sugar libraries without any limitation on the number of metalevels involved. To exemplify this point, we have embedded XML Schema declarations into SugarJ as a sugar library for defining sublanguages of XML. XML specification languages such as XML Schema [45] serve the purpose of declaring such subsets of XML and can be used to check whether a given XML document is *valid*. Therefore, each concrete XML specification stipulates a DSL of valid XML documents; a language of XML specifications is a meta-DSL.

To validate XML documents through the compiler, we have integrated a subset of XML Schema into SugarJ by implementing a sugar library. As shown in Fig. 10(a), a programmer can define an XML schema using a top-level

```
import xml.schema.XmlSchema;

public xmlschema BookSchema {
  <{http://www.w3.org/2001/XMLSchema}schema
      targetNamespace="lib">
    <!-- define schema content here -->
  </{http://www.w3.org/2001/XMLSchema}schema>
}
```

(a) Definition of an XML schema for the namespace lib.

```
import xml.XmlJavaSyntax;
import xml.AsSax;
import BookSchema;

public void genXML(ContentHandler ch) {
  @Validate
  ch.<{lib}book title="Sweetness and Power">
      <{lib}author name="Sidney W. Mintz">
      </{lib}author>
    </{lib}book>;
}
```

(b) SugarJ statically validates XML documents whenever validation is required by the "write valid document" statement. To relate XML elements to their schema definition, element names are quantified with namespaces, here {lib}.

Figure 10. Definition and application of an XML schema.

xmlschema declaration that contains a conventional XML Schema document.[4] A programmer can require the validation of an XML document by annotating it using @Validate, as we illustrate in Fig. 10(b). During compilation, the XML schema of the corresponding namespace traverses the XML document to check its validity and generate a possibly empty list of error messages.

Technically, we have defined a program transformation that desugars an XML schema into transformation rules for validating XML documents. An XML Schema element declaration

```
<{http://www.w3.org/2001/XMLSchema}element
    name="book" type="BookType">
</{http://www.w3.org/2001/XMLSchema}element>,
```

for example, desugars into a program transformation that matches on XML elements book and checks whether their attributes and children conform to BookType. According to the structure of an XML schema, validation rules like this one are composed to form a full validation procedure for matching XML documents and collecting possible errors. The XML Schema sugar library tries to validate an XML document against any validation procedure that is in scope.

[4] For simplicity, we currently neither support namespace abbreviations xmlns:abc="xyz" nor empty elements <author name=".."/> for XML Schema. However, these features are syntactic sugar and can easily be implemented in additional sugar libraries.

Should no schema exist for the XML document's namespace, the sugar library issues a corresponding error message.

The XML Schema case study not only demonstrates SugarJ's support for compile-time checks, but moreover its self-applicability support: The sugar library introduces syntactic sugar (XML Schema declarations) for the specification of metaprograms. This possibility of applying SugarJ to itself allows programmers to build meta-DSLs.

SugarJ's extensive support for self-application was also helpful in our implementation of the XML Schema sugar library itself. Although standard XML Schema cannot describe itself in general [29], we identified a self-describable subset of the language. This allowed us to bootstrap the sugar library for XML Schema declarations from a description of its syntax as an XML Schema declaration.

In summary, we have presented five case studies showing the expressiveness and applicability of SugarJ for implementing language extensions and syntactically embedding DSLs. Especially the more complex sugar libraries reuse simpler libraries, and with XML Schema we demonstrate SugarJ's flexibility as well as the benefits of context-sensitive checks and self-application.

6. Discussion and future work

In the present section, we discuss SugarJ's current standing, its limitations, and possible future development with respect to language composability, context-sensitive checks, tool support and a formal consolidation.

6.1 Language composability

In contrast to previous language extensibility approaches, composing languages with SugarJ is very simple since it only involves importing sugar libraries. However, when composing multiple DSLs, ambiguities can arise in composed grammars and desugaring rules, or additional glue code might be necessary to integrate both languages more carefully (introduce intended interactions and prevent accidental interactions).

Nonetheless, when composing language extensions, our experience with SugarJ suggests that ambiguity problems do not occur frequently in practice or are easily resolvable. For instance, no composition problems arise in the case studies presented in Section 5.

In the general case, though, syntactic conflicts can certainly arise, which are detected at compile-time. For instance, when composing our XML sugar library with a library for HTML, the parser will recognize a syntactic ambiguity in the following program, because the generated document could be part as either language:

```
import Xml;
import Html;

public void genDocs(ContentHandler ch) {
  ch.<book title="Sweetness and Power">
      <author name="Sidney W. Mintz" />
    </book>;
}
```

If a conflict like this arises, it is always possible to resolve conflicts without changing the composed sugar libraries. For instance, one can add an additional syntax rule which allows the user to write, say, ch.xml<...> or ch.html<...> to resolve the disambiguity.

Another potential composition problem arises when importing multiple desugarings for the same extended syntax. The compiler will not detect the resulting conflict in the desugaring rules, leading to unexpected compile-time errors during desugaring. We believe that this is again not a big practical problem for the scenario of syntactic sugar for a DSL, since usually each DSL comes with its own syntax, and hence the desugaring rules do not overlap.

That said, detecting syntactic and semantic ambiguities or conflicts is an interesting research topic related to feature interactions [9]. While not in the scope of this work, in our future work we plan to evaluate existing technologies for detecting ambiguities in grammars and program transformations. For example, we want to investigate the applicability of Axelsson *et al.*'s encoding of context-free grammars as propositional formulas, which allows them to apply SAT solving to verify the absence of ambiguous words up to a certain length in short time, but may fail to terminate in the general case [2]. Alternatively, Schmitz proposed a terminating algorithm that conservatively approximates ambiguity detection for grammars and generalizes on the ambiguity check build into standard LR parse table construction algorithms [35]. For the detection of conflicting desugaring rules, we want to assess the practicability of applying critical pair analysis to prohibit all critical pairs—even joinable ones—reachable from the entry points of desugaring. This idea has previously been applied for detecting conflicts in program refactorings [27]. To rule out fewer critical pairs, we could combine critical pair analysis with automatic confluence verification [1] to determine the joinability of critical pairs.

In contrast to most other metaprogramming systems, the ambiguity detection algorithms described above could conceptually all be implemented within SugarJ as metalanguage compile-time checks. Since these checks would operate on the fully desugared base language, however, SugarJ would need to support more fine-grained control over *when* checks are executed.

6.2 Expressiveness of compile-time checks

Sugar libraries natively support checking user code for syntactic and semantic correctness, as each syntactic extension specifies what correctness means in terms of a context-free grammar and compile-time assertions. During parsing, conformance to an extension's grammar is checked. For example, we ensure matching brackets in our pair and closure DSLs.

For context-sensitive properties (necessary, for example, for context-sensitive languages or statically typed DSLs), however, the question arises when to check them. In addition to encoding constraints as part of desugaring rules, our current implementation of SugarJ also offers initial support for a more direct implementation of error reporting: Sugar libraries can specify a Stratego transformation which transforms the non-desugared syntax tree of an input file into a list of error messages. This approach allows the definition of context-sensitive properties in terms of surface syntax and comprises pluggable type systems [6]. For instance, the check for matching start and end tags of XML documents and XML Schema validation is naturally specified in terms of XML syntax. However, the use of non-desugared syntax restricts the extensibility of compile-time checks. Consider, for example, a syntactic extension that introduces JavaScript Object Notation (JSON) syntax as an alternative syntax for describing tree-structured data, which desugars to XML code:

```
{
  "book":{
    "title" : "Sweetness and Power",
    "author" : {"name":"Sidney W. Mintz"}
  }
}
```

Even though this code desugars to XML code eventually, our current implementation of XML Schema validation will fail because it operates on the non-desugared JSON syntax tree, but can only match on XML documents. Similar to the ambiguity checks described in the previous subsection, we here require some interleaving of compile-time checking and desugaring to enable compile-time checks on surface syntax, base language syntax and intermediate syntax. To this end, in our future work, we would like to investigate the applicability of a constraint system that separates constraint generation from constraint resolution and performs both interleaved with desugaring. We plan to let constraints keep track of the actually performed desugarings so that constraint verification does not interfere with the application of desugarings.

6.3 Tool support

In order to efficiently develop software in the large, error reporting, debugging and other IDE support is essential [16, 22, 34]. Due to the fluent change of syntax, and thus language, sugar libraries place extraordinary challenges on tools: all language-dependent components of an IDE depend on the sugar libraries in scope. Consider syntax highlighting, for example, in which keywords are colored or highlighted in a bold font. Since syntactic extensions can introduce new keywords to the host language, syntax highlighting needs to take sugar-library imports into account.

In fact, we are currently working on an integration of SugarJ and Spoofax. Spoofax provides a framework for developing IDEs by specifying a language's syntax and language-specific editor services such as reference resolving and content completion [22]. From these specifications, Spoofax generates a fully-fledged, Eclipse-based editor component for the user's language that can be hot-swapped while the user types. Depending on the imported sugar libraries, we plan to automatically generate and reload editors for each source file independently, so that an editor corresponding to a file always reflects the language extensions in scope. While basic editor services such as syntax highlighting are easy to integrate by reusing syntax definitions from sugar libraries, more sophisticated services require the implementation of language-dependent analyses. We propose to implement these analyses in *editor libraries*, which in conjunction with a language's sugar library supplies the necessary information for providing advanced editor services in a library-centric fashion automatically.

6.4 Core language

In the study of sugar libraries, we used SugarJ to evaluate the expressiveness and applicability of our approach, for instance, by developing complex case studies such as XML Schema. However, it would be interesting to formally consolidate sugar libraries and study them more fundamentally.

One aspect we intend to study is the relation between syntactic extensions and scopes. It is not obvious how to support sugar libraries in languages that allow "local" import statements, e.g., within a method, such as in Scala and ML. Consider for instance the following piece of code, in which we assume `s1 after s2` to desugar to `s2; s1`, i.e., to swap the order of the statements `s1` and `s2`.

```
("12",34) after import pair.PairSugar
```

After swapping the two statements, the scope of the import of `pair.PairSugar` includes `("12",34)`, which, thus, is a syntactically valid expression. However, to parse a program in the form `s1 after s2`, the parser already requires knowledge of how to parse `("12",34)` before it can even consider parsing `import pair.PairSugar`; this is a paradox.

Another interesting aspect of such a core language is to identify the minimal components of a syntactically extensible language such that a full language like SugarJ can be boot-strapped from this core language.

6.5 Module system

The semantics of imports in SugarJ is intended to closely match the semantics of imports in Java. In our proof-of-concept implementation, however, imports are split into Java, SDF and Stratego by reproducing them in the respective syntax. Unfortunately, though, the scoping rules of these languages differ: Imports are transitive in Stratego and SDF but non-transitive in Java. Therefore, in the current implementation of SugarJ, if `A` imports syntactic sugar from `B`, which in turn imports syntactic sugar from `C`, the syntactic sugar from `C` will be available in `A`. In contrast, `A` cannot

access Java declarations from C without first importing C or using fully qualified names. We want to investigate whether this mismatch can be resolved using some form of systematic renaming in our future work.

Java, the base language for SugarJ, has a rather simple module system in which the interface of a library is often rather implicit because users of a library just import the library's implementation.

In our future work we would like to make syntactic extensions a formal part of a dedicated interface description language. In this context we also want to address the question of whether there should be some kind of abstraction barrier in an interface that hides the details of the desugaring of a syntactic extension. In the current SugarJ programming model, a programmer has to understand the associated desugaring to reason about, say, the well-typedness of a program written in extended syntax. Hence the desugaring rules must be part of the interface. We believe that this is acceptable as long as transformations are simple and compositional – which is typically the case for syntactic sugar. However, if more sophisticated transformations would be performed, it would make sense to have an abstraction mechanism that hides the details of the transformation, yet allows programmers to reason about their code in terms of the interface.

7. Related work

In Table 1, we give an overview that compares syntactic embedding approaches regarding the novel combination of features of sugar libraries. Sugar libraries provide a (1) metalevel-agnostic library-based approach for implementing (2) arbitrary domain-specific syntax extensions in a (3) composable and reusable form. Furthermore, sugar libraries arrange to (4) statically check user code at compile-time and (5) act on all metalevels in a self-applicable fashion.

String encoded DSLs require the escaping of quotes and only allow lexical composition of programs, that is, by string concatenation. Furthermore, string encoded programs are not statically checked but parsed at runtime.

In contrast, Hudak's *pure embedding* [21] does not suffer from the deficits of string encodings. However, since in a pure DSL embedding all implementation aspects are encoded into the host language, domain syntax and static checking are restricted through the host language's syntactic flexibility and type system. Therefore, a pure embedding of XML with domain syntax and document validation as in SugarJ remains a distant prospect.

Macro systems such as Scheme, the C preprocessor, C++ templates, M4, and Dylan (see their comparison by Brabrand and Schwartzbach [5]), as well as the Java syntactic extender [3] and the macro-like metaprogramming facilities of Converge [40] support user-defined syntax, but user-defined syntax is restricted to macro arguments [5]. Macros systems are either lexical or syntactic. Lexical macro systems such as the C preprocessor perform program transformations oblivious to the program's structure and are therefore ill-suited for implementing DSL desugarings. Syntactic macro systems such as Scheme, on the other hand, parse a program before expanding macros in the abstract syntax tree.

Scheme macros are organized in libraries [15] and provide referentially transparent and hygienic structural rewriting of programs [11, 24], that is, macro expansion respects the lexical scoping of identifiers. While sugar libraries also respect referential transparency when using fully qualified class names in desugarings, we opted for more flexible but unhygienic transformations in our prototype implementation. For instance, the composability of Scheme macros is limited, since they prohibit macro expansion in quoted expressions such as the patterns of a case statement [23, 24]. The design of a flexible and extensible (including the definition of new variable binders) mechanism for keeping desugarings hygienic and referentially transparent would be an interesting avenue of future work.

Recently, Tobin-Hochstadt *et al.* proposed Racket, a dialect of Scheme, as a host language for library-based language extensibility [39]. In contrast to Scheme and similar to Lisp, Racket provides facilities for adapting its lexical syntax (using readers) and thus supports more flexible syntactic embeddings of DSLs. However, Racket lacks support for a high-level syntax formalism, and reader implementations do not compose well. While SugarJ addresses language extensions of varying complexity through composition, Tobin-Hochstadt *et al.* focus on singular large-scale language extensions; it is not clear how to compose languages in Racket.

The flexibility of language extensions through *extensible compilers*, in general, provides excellent support for syntactic embedding of domain-specific syntax into a host language. Unfortunately, the host/base language of extensible compilers is usually different from their metalanguage. For example, none of the extensible Java compilers JastAddJ [13], AbleJ [42] and Polyglot [30] uses Java as their metalanguage; they provide their own specification languages instead. Consequently, application developers cannot select and develop language extensions in their programs, but must declare and define the desired extensions externally. Furthermore, in implementing language extensions, programmers cannot employ already existing syntactic extensions because the extended language is different from the metalanguage. Finally, the mentioned extensible compilers rely on syntax formalisms that are not closed under the composition of grammars and therefore suffer syntactic composability issues.

In contrast to extensible compilers, *program transformation systems* such as MetaBorg [8] and Silver [41] support self-applicability, that is, it is possible to write transformations for extending the system itself (indeed, Silver itself is implemented in Silver). In these systems, programmers can write transformations that target the system itself, thus enabling the definition of meta-DSLs. However, to com-

Approach	Libraries across metalevels	Domain syntax	Composability	Static checking	Self-applicability
string encoding	●	◒	◒	○	●
pure embedding [21]	●	○	●	◒	●
macro systems [3, 5, 39, 40]	●	◒	◒	●	●
extensible compilers [13, 30, 42]	○	●	◒	●	○
prog. transformations [8, 41]	○	●	●	●	◒
dynamic meta-object protocols [33, 34]	○	●	○	n/a	●
sugar libraries in SugarJ	●	●	●	●	●

● addressed as goal ◒ addressed but with restrictions ○ not regarded as goal

Table 1. Comparison of syntactic DSL embedding approaches.

pile a meta-DSL program, a programmer needs to execute three steps: first, compile the meta-DSL transformation using the base compiler; second, apply the meta-DSL transformation to the meta-DSL program; third, apply the base compiler to the transformed meta-DSL program. For more advanced uses of DSLs and meta-DSLs, as, for example, in our XML Schema case study, performing the relevant compilation steps by hand is hardly practical and requires tool support in form of build scripts. In contrast, a SugarJ programmer never has to leave the programming language and can compile all programs alike, namely by calling the SugarJ compiler only once.

Meta-object protocols enable the implementation of language extensions through reflection and modification of objects and classes at either compile-time or runtime. Static meta-object protocols such as OpenJava [38] are similar to macro systems and focus on semantic extensibility but provide limited support for syntactic extensions. *Helvetia*, on the other hand, is a reflection-based extensible compiler and IDE for the Smalltalk programming language that supports syntactic extensibility through a dedicated *dynamic meta-object protocol* [33, 34]. In Helvetia, syntactic extensions are implemented as Smalltalk programs that parse and transform other Smalltalk programs. These metaprograms, however, cannot be composed and are not organized as libraries—they are not imported but rather activated within a Smalltalk image. Similar to Helvetia, Katahdin [36] supports dynamic syntax adaption at runtime. However, the author does not discuss the composability and self-applicability of language extensions in Katahdin.

Finally, *projectional workbenches* [12, 16, 37] add an interesting twist, by working around parsing entirely. Instead of parsing source code, projectional workbenches store programs in a database and, on demand, provide projections in textual form. Programmers change programs in structure editors, some of which emulate textual editors to some degree. Projectional workbenches allow easy extension and composition of languages (by extending and composing their underlying structures) and support self-applicability in projections. Since syntax is projected from an underlying structure, it is not even restricted to textual representations. Since projectional workbenches work with a very different model of source code, they cannot be directly compared to SugarJ, which focuses on traditional text-based source code.

8. Conclusion

We introduced sugar libraries as a modular mechanism to extend a host language with domain-specific syntax. Developers can import syntax extensions and their desugaring as libraries, for instance, to develop statically checked domain-specific programs. Sugar libraries preserve the look-and-feel of conventional libraries and facilitate composability and reuse: A developer may flexibly select from multiple syntactic extensions and import and combine them, and a library developer may reuse sugar libraries when developing other sugar libraries (even in a self-applicable fashion). Composition conflicts can occur, but we believe that they are rare in practice. We would still like to have better support for avoiding (by better scoping constructs) and detecting (by better analyses) composition conflicts statically.

To demonstrate flexibility and expressiveness, we have implemented sugar libraries in the Java-based language SugarJ. With SugarJ, we have implemented five case studies with growing complexity: pairs, closures, XML, concrete syntax for transformations and XML Schema. The latter of these case studies heavily reuse syntax extensions imported from former and the last one implements a meta-DSL for which self-applicability is a significant advantage. In contrast to many other metaprogramming systems, a SugarJ programmer never has to reason outside the language since SugarJ comprises full metaprogramming facilities.

Our main goal for SugarJ is to provide a library-based mechanism for developing and embedding domain specific languages. More generally, we believe that libraries are the best means for organizing software artifacts into reusable and composable units, and should be more often utilized to account for the growing complexity of software systems.

Acknowledgments

We would like to thank Lennart Kats, Eelco Visser, Shriram Krishnamurthi and Ralf Lämmel for discussions on this project. This work is supported in part by the European Research Council, grant No. 203099.

References

[1] T. Aoto, J. Yoshida, and Y. Toyama. Proving confluence of term rewriting systems automatically. In *Proceedings of Conference on Rewriting Techniques and Applications (RTA)*, LNCS, pages 93–102. Springer, 2009.

[2] R. Axelsson, K. Heljanko, and M. Lange. Analyzing context-free grammars using an incremental SAT solver. In *Proceedings of International Colloquium on Automata, Languages and Programming (ICALP)*, LNCS, pages 410–422. Springer, 2008.

[3] 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.

[4] D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: Tools for implementing domain-specific languages. In *Proceedings of International Conference on Software Reuse (ICSR)*, pages 143–153. IEEE Computer Society, 1998.

[5] C. Brabrand and M. I. Schwartzbach. Growing languages with metamorphic syntax macros. In *Proceedings of Workshop on Partial Evaluation and Program Manipulation (PEPM)*, pages 31–40. ACM, 2002.

[6] G. Bracha. Pluggable type sysmtes. In *OOPSLA Workshop on Revival of Dynamic Languages*, 2004.

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

[8] 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.

[9] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec. Feature interaction: A critical review and considered forecast. *Computer Networks*, 41(1):115–141, 2003.

[10] L. Cardelli. Program fragments, linking, and modularization. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*, pages 266–277. ACM, 1997.

[11] W. Clinger and J. Rees. Macros that work. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*, pages 155–162. ACM, 1991.

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

[13] 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.

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

[15] M. Flatt. Composable and compilable macros: you want it when? In *Proceedings of International Conference on Functional Programming (ICFP)*, pages 72–83. ACM, 2002.

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

[17] N. Gafter and P. von der Ahé. Closures for java. Available at `http://javac.info/closures-v06a.html`. Accessed April 5, 2011.

[18] 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.

[19] C. Hofer and K. Ostermann. Modular domain-specific language components in Scala. In *Proceedings of Conference on Generative Programming and Component Engineering (GPCE)*. ACM, 2010.

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

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

[22] 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.

[23] O. Kiselyov. Macros that compose: Systematic macro programming. In *Proceedings of Conference on Generative Programming and Component Engineering (GPCE)*, volume 2487 of *LNCS*, pages 202–217. Springer, 2002.

[24] E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba. Hygienic macro expansion. In *Proceedings of Conference on LISP and Functional Programming (LFP)*, pages 151–161. ACM, 1986.

[25] S. Krishnamurthi. Educational pearl: Automata via macros. *Journal of Functional Programming*, 16(3):253–267, 2006.

[26] B. M. Leavenworth. Syntax macros and extended translation. *Communications of the ACM*, 9:790–793, 1966.

[27] T. Mens, G. Taentzer, and O. Runge. Detecting structural refactoring conflicts using critical pair analysis. *Electronic Notes in Theoretical Computer Science*, 127(3):113–128, 2005.

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

[29] A. Møller and M. I. Schwartzbach. *An Introduction to XML and Web Technologies*. Addison-Wesley, 2006.

[30] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for Java. In *Proceedings of*

Conference on Compiler Construction (CC), pages 138–152. Springer, 2003.

[31] M. Odersky. The Scala language specication, version 2.8. Available at `http://www.scala-lang.org/docu/files/ScalaReference.pdf`., 2010.

[32] B. C. Oliveira. Modular visitor components. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, pages 269–293. Springer, 2009.

[33] L. Renggli, M. Denker, and O. Nierstrasz. Language boxes: Bending the host language with modular language changes. In *Proceedings of Conference on Software Language Engineering (SLE)*. Springer, 2009.

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

[35] S. Schmitz. Conservative ambiguity detection in context-free grammars. In *Proceedings of International Colloquium on Automata, Languages and Programming (ICALP)*, LNCS, pages 692–703. Springer, 2007.

[36] C. Seaton. A programming language where the syntax and semantics are mutable at runtime. Master's thesis, University of Bristol, 2007.

[37] C. Simonyi. The death of computer languages, the birth of intentional programming. In *NATO Science Committee Conference*, 1995.

[38] 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.

[39] 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. to appear.

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

[41] 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.

[42] 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)*, LNCS, pages 575–599. Springer, 2007.

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

[44] E. Visser. Meta-programming with concrete object syntax. In *Proceedings of Conference on Generative Programming and Component Engineering (GPCE)*, LNCS, pages 299–315. Springer, 2002.

[45] W3C XML Schema Working Group. XML schema part 0: Primer second edition. Available at `http://www.w3.org/TR/xmlschema-0`, 2004.

[46] W3C XML Working Group. Extensible markup language (XML) 1.0 (fifth edition). Available at `http://www.w3.org/TR/xml`, 2008.

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

[48] 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.