Designing Syntax Embeddings and Assimilations for Language Libraries

Martin Bravenboer and Eelco Visser

Software Engineering Research Group, Delft University of Technology,
The Netherlands, martin@st.ewi.tudelft.nl, visser@acm.org

Abstract. *Language libraries* extend regular libraries with domain-specific notation. More precisely, a language library is a combination of a domain-specific language *embedded* in the general-purpose host language, a regular library implementing the underlying functionality, and an *assimilation* transformation that maps embedded DSL fragments to host language code. While the basic architecture for realizing language libraries is the same for all applications, there are many design choices to be made in the design of a particular combination of library, guest language syntax, host language, and assimilation. In this paper, we present a systematic analysis of the design space for syntax embeddings and assimilations for the realization of language libraries. The contribution of this paper is an overview of the state-of-the-art providing insight in the design space and research questions in language library realization, in particular, the identification of research issues for realizing an independently extensible language library framework.

1 Introduction

Software libraries provide reusable data structures and functionality through the built-in abstraction facilities of a programming language. While functionally complete, the interface through regular function or method calls is often not appropriate for efficiently and understandably expressing programs in the domain of the library. *Language libraries* extend regular libraries with domain-specific notation. More precisely, a language library is a combination of a domain-specific language *embedded* in the general-purpose host language, a regular library implementing the underlying functionality, and an *assimilation* transformation that maps embedded DSL fragments to host language code.

Prominent applications of language libraries are found in *meta-programming* where the objects of computation are programs, which are naturally represented through their concrete syntax. In program transformation, structured representations of programs are modified. In generation, programs are composed by instantiating code templates. Another application of language libraries is the generation of programs in languages such as SQL and regular expressions, which are supported by libraries taking as arguments strings composed by unsafe string concatenation. A final area of application is providing domain-specific notation for existing libraries through a domain-specific language [11].

Language libraries are realized by means of *syntax embeddings and assimilations* as illustrated by the architecture diagram in Figure 1. An implementation typically consists of four components, *i.e.* a parser, typechecker, assimilator, and pretty-printer. Together these components transform programs in the extended language to programs in the host language only. Each of the components is parameterized with data that are specific for the syntax embedding at hand. The *parser* is parameterized with the syntax of the extended language combining the host language and the guest language. This requires syntax definitions for the *host language* and the *guest language*, and a definition of how guest language fragments are inserted in host language programs. The parser converts a textual representation of a program in the combined language to a tree structure that is suitable for further processing. *Type rules* extend the *typechecker* of the host language to check extended programs. This component is optional. Having an extensible typechecker avoids type error messages expressed in terms of assimilated programs. The *assimilator* transforms embedded guest language fragments to an implementation in the host language. The

assimilator is parameterized with a set of *assimilation rules* that define the translation schemes for the guest language. For certain applications the assimilation rules may be generic in the guest language. A *pretty-printer* converts the tree structure produced by the assimilator to text, which can be fed to a host language compiler or interpreter. Pretty-printing can be avoided if transformations are expressed directly on parse trees, rather than abstract syntax trees. Another option is not to produce a textual representation of the assimilated program at all, but instead link assimilation into the host language compiler. This is done for example in Stratego, where assimilation of concrete object syntax is built into the compiler [25]. Finally, the assimilator may generate code that makes calls to an *API corresponding to the guest language* (the 'run-time system' for the embedding).

In recent years case studies of language libraries have been conducted for a number of host languages and types of applications, including concrete object syntax for meta-programming [3, 25], embedding of domain-specific languages in general purpose languages (MetaBorg) [11, 7], and syntax embeddings for preventing injection attacks (StringBorg) [8]. While the architecture outlined above applies to all these language libraries, there are many design choices to be made in filling in the parameters to the architecture. For example, a recent innovation is type-based disambiguation of syntax embeddings [27, 10], which uses the type system of the host language to disambiguate quoted code fragments, thus allowing a more lightweight syntax.

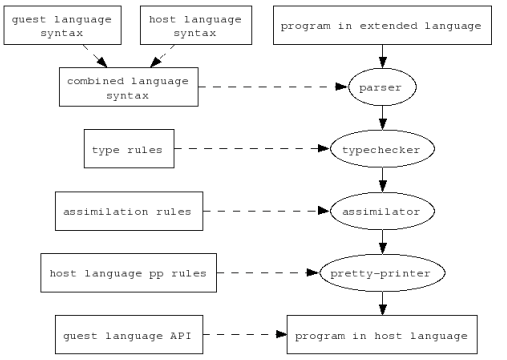

Fig. 1. Components for syntax embedding and assimilation. Solid arrows denote data-flow, dashed arrows denote parameters of tools in the chain.

In this paper, we present a systematic analysis of the design space for syntax embeddings and assimilations for the realization of language libraries. The contribution of this paper is an overview of the state-of-the-art providing insight in the design space, and research questions in language library realization, in particular, the identification of research issues for realizing an independently extensible language library framework. In the next section we give an overview of the various categories of language libraries illustrated with examples. In the remaining sections we discuss technical considerations and trade-offs in the realization of language libraries.

2 Language Library Categories

In this section we consider different types of applications, which are characterized by the target of assimilation. We distinguish four types of language libraries; libraries for transformation, generation, string generation, and DSL embedding. We consider each of these categories in turn and show typical examples. Figure 2 gives an overview of some libraries that we have realized.

Transformation of structured data *Transformation of structured data* is typically used for program transformation, but also for transformation of other structured data, such as XML documents. Direct manipulation of the structures through their API can lead to tedious coding in which the natural structures are hard to recognize. Syntax embeddings can be used to provide concrete syntax for patterns used to match and construct code fragments [25]. The target of assimilation in these applications is an API for analysing and constructing abstract syntax trees. For example, consider the following Stratego rewrite rule that defines the desugaring of the `sum` construct in terms of a `for` loop with an auxiliary (fresh) variable:

```
DefSum :
  |[ sum x = e1 to e2 ( e3 ) ]| ->
  |[ let var y := 0 in (for x := e1 to e2 do y := y + e3); y end ]|
  where y := <newname> x
```

The terms between `|[` and `]|` are quotations of Tiger code patterns that are used both to pattern match *and* to compose code. For example, the left-hand side `|[ sum x = e1 to e2 (e3) ]|` of the rewrite rule is assimilated to the term pattern `Sum(Id(x), e1, e2, e3)`, where `x`, `e1`, `e2`, and `e3` are *meta-variables*, i.e. variables that match subterms when the rule is applied. (`newname` is used to create a fresh variable in order to avoid accidental capture of free variables.)

A similar idea can be used with Java as host language. While Stratego natively supports terms for representing abstract syntax trees, Java requires such structures to be defined using objects. A syntax embedding of terms in Java (JavaATerm) can be used to make analysis and construction of term structures in Java programs easier. For example, the following is a code fragment from a transformation from an ATerm representation of bytecode files to a BCEL representation of bytecode:

```
private void addInstructions(ATerm code) {
  ATerm optlocals = null, optstack = null;
  ATerm is = null, excs = null;
  match code with
    Code(MaxStack(optstack), MaxLocals(optlocals)
    , Instructions([...] and is)
    , ExceptionTable([...] and excs)
    , Attributes(_));
```

The embedding provides (among other things) the `match <expr> with <term>` construct and concrete syntax for term patterns. The construct is assimilated to statements that implement the matching by analyzing the ATerm argument against the declared pattern.

	host language			
guest lang.	Stratego	Java	Perl	PHP
Stratego	T			
Tiger	T			
ATerm	T	T		
C	G(T)			
Java	G(T)	G		
XML	G(T)	G		
SQL		S	S	S
Shell		S	S	S
XPath		S	S	S
Swul		D		
Regex		D		

Fig. 2. Examples of realized embeddings with host languages in columns and embedded languages in rows. The letters indicate the type of embedding with G = generation, T = transformation, S = string generation, D = DSL embedding.

An important requirement for the use of syntax embeddings in transformations is that the structure of the quoted pattern coincides with the structure of the program to which it is applied. This does not hold in scenarios where the abstract syntax tree of a program is heavily analyzed, modified, and/or attributed before being transformed. For example, abstract syntax trees for C and Java require static semantic analysis before they can be properly transformed.

Code generation *Code generation* involves the composition of programs from templates based on input such as a DSL program or user input. Construction of large pieces of code using string concatenation is tedious and error-prone. Templates are not checked statically for well-formedness and meta characters need to be escaped. Furthermore, a textual representation does not allow further processing. Use of an API provides a structured representation, but is not suitable for encoding (large) templates. Syntax embeddings allow encoding templates in the concrete syntax of the language, but at the same time producing structured code. Embedded code fragments are assimilated to API calls for *constructing* structured data (such as ASTs). The API does not need to support *transformation* of patterns derived from concrete syntax. For example, the back-end of the Stratego compiler uses rules such as

```
TranslateStrat(|S,F) :
  |[ s1 < s2 + s3 ]| ->
  stm|[ { ATerm ~id:x = t;
          ~stm:<translate-strat(|Next,F')>s1
          ~stm:<translate-strat(|S,F)>s2
          ~id:F' : t = ~id:x;
          ~stm:<translate-strat(|S,F)>s3 } ]|
  where x := <new>; F' := <new>
```

to implement schemes for translating Stratego to C. In this example *two* languages are embedded in the host language; the left-hand side of the rule is a Stratego code fragment, while the right-hand side is a C code fragment. The right-hand side template uses recursive calls to generate code for the subterms of the left-hand side pattern. The results are integrated by *anti-quotations*

such as `~stm:t` and `~id:t` that convert a Stratego term into the right syntactic sort. Note that quotations and antiquotations are *tagged* with syntactic sorts such as `stm|[` and `~id:`, this is necessary to avoid ambiguities in parsing.

The same technique is used for other host languages. For example, the following fragment shows a Java method generating Java code represented using the Eclipse JDT:

```
public CompilationUnit run(ClassDescriptor cd) {
  CompilationUnit result = |[
    import java.util.*;
    public class #[ cd.getName() ] {
      #[ attributes(cd) ]
    }
  ]|;
  return result;
}
```

Note that unlike in the Stratego case, the (anti)quotations are not tagged; such syntactic information can be deduced from the types of the host language (e.g., `CompilationUnit`) [10].

String generation It is common practice for libraries supporting domain-specific languages such as XML, XPath, regular expressions, and SQL to provide an interface that uses character strings to communicate programs in these language. For example, a method `execQuery` from an SQL library might take a string containing an SQL query as argument, such as

```
execQuery("SELECT * FROM Users where username = \'" + name + "\'");
```

Insertion of run-time data (e.g., user input) is done by composing queries using string concatenation from constant parts and dynamic parts provided by variables. The approach suffers from a number of problems. First, strings are not statically guaranteed to be well-formed, which may lead to run-time errors. Second the approach requires escaping of meta-symbols (such as the quotes in the example above), which can lead to tedious and obfuscated code. Worst of all, the approach leads to software that is prone to *injection attacks*, where ill-formed user input may lead to breakdown or security compromises. For example, insertion of the string `' OR 1=1--` in the query above would lead to dumping a list of all users.

These problems can be avoided by using syntax embeddings instead of string concatenation. By quoting a query, as in the following code fragment

```
SQL.QueryExpr q = <| SELECT * FROM Users WHERE username = $str{arg} |>;
execQuery(q);
```

the query is syntactically checked at compile-time, no spurious escaping is needed, and insertion of run-time data is guaranteed to be properly escaped to avoid injection attacks. Embedded queries are assimilated to calls to methods in a *syntax API*, which provides for each production in the syntax definition a corresponding method. This ensures that well-formedness of queries is verified by the type system of the host language.

DSL embedding The final category of language libraries we consider is those of domain-specific language (DSL) embeddings. While the previous categories were concerned with transformation or generation of programs, DSL embeddings are concerned with providing better notation for programming in a certain domain, and are typically designed around an existing library. The DSL abstracts over normal usage of the library, and assimilation is to sequences of library calls. For example, JavaRegExp is a DSL built on top of the Java regular expression library. In the first place it provides a syntax for quoting regular expressions without spurious escaping, similar to the string generation examples above. Building on this basic embedding, JavaRegExp provides a DSL for defining and combining string rewrite rules in Java. For example, the following code fragment defines several string rewrite rules for escaping HTML special characters, their composition with a choice operator `<+`, and the application of the rules to the contents of a string in variable `input`:

```
regex amp = [/ & /] -> [/ &amp; /];
regex lt  = [/ < /] -> [/ &lt;  /];
regex gt  = [/ > /] -> [/ &gt;  /];
regex escape = amp <+ lt <+ gt;
input ~= all(escape);
```

An issue in the design of JavaRegExp is the identification of DSL objects such as a string rewrite rule, since these are compile-time objects only, with no counterparts in the regular expresssion library.

Another example in this category is JavaSwul, an embedding in Java of a dedicated language for creating Swing user-inteface widgets, following the hierarchical structure of the class hierarchy of Swing [11, 7].

3 Syntax Embedding

In this section, we discuss the design of syntax embeddings. The basic idea of a syntax embedding is to extend the syntax of the host language with the syntax of a guest language. Such an extension is achieved by a modular extension of the grammar of the host language, which introduces the language constructs of the guest language at specific places in the host language. In this method for syntax embedding, there are several design decisions to make. First, the syntax for the transition from the host to the guest language and back needs to be designed. Second, if necessary, a method for dealing with ambiguities needs to be chosen. Third, an additional keyword policy might be useful for a particular combination of two languages. Finally, several parsing methods exist that are capable of dealing with syntax embeddings.

```
module TinySQL exports
lexical syntax
  [A-Za-z]+ -> Id                                    1
  [A-Za-z0-9\ \"\-\;] -> Char
  "\'" ("\'\'" | Char)* "\'" -> CharString           2
context-free syntax
  "SELECT" Id* "FROM" Id Where?-> Query              3
  "WHERE" Expr -> Where                              4
  Expr "=" Expr -> Expr {left}                       5
  CharString -> Expr
  Id -> Expr
```

Fig. 3. Syntax definition for a tiny subset of SQL

3.1 Introducing Syntax Embedding

Before we discuss the details of the design of syntax embeddings, we present a syntax embedding of SQL in PHP, to illustrate the basic implementation. For the examples we use SDF [24], a modular syntax definition formalism for defining the lexical as well as context-free syntax of language in a single formalism. We will elide the details of the embedding, to return to these later.

A syntax embedding requires syntax definitions for the host and guest languages. Figure 3 shows the syntax definition for a tiny subset of SQL, the guest language of this example. Naturally, we omit the syntax definition of the host language. The module `TinySQL` defines the lexical and context-free syntax of a stylized subset of SQL in reversed EBNF notation, namely simple queries [3] with where clauses [4], equivalence expressions [5], identifiers [1] and character strings with escaped quotes [2].

The syntax of TinySQL is embedded in PHP by creating a new SDF module (Figure 4) that imports the syntax definitions of PHP as well as TinySQL [6] and defines how the languages are combined, *i.e.* where TinySQL can be used in

```
module SQL-in-PHP imports PHP TinySQL [6] exports
context-free syntax
  "<|"    Query⟦SQL⟧ "|>" -> Expr⟦PHP⟧          7
  "${"    Expr⟦PHP⟧  "}"  -> Expr⟦SQL⟧          8
  "$str{" Expr⟦PHP⟧  "}"  -> CharString⟦SQL⟧    9
```

Fig. 4. Embedding of syntax for SQL in syntax for Java.

PHP and vice versa. The productions of this module use *parameterized non-terminals*, *e.g.* `Expr[[PHP]]` and `Expr[[SQL]]`, which are used to indicate the language of the non-terminal. We will discuss the exact origin and meaning of these parameters later. The first production [7] of `SQL-in-PHP` defines the *quotation* of SQL queries. SQL queries can be used as PHP expressions between the quotation tokens `<|` and `|>`. The second [8] and third [9] productions define the *anti-quotations* of this embedding, which allow an SQL expression or character string to be constructed by an arbitrary PHP expression. In this way, queries can be composed dynamically.

From this combined syntax definition a parser is generated, which is used to parse PHP programs that use the SQL syntax extension. After that, the resulting parse tree or abstract syntax tree is transformed to a plain PHP program by an assimilation. Assimilations will be discussed in Section 5, so we ignore the further processing of the program for now.

3.2 Combining Host and Guest Languages

The syntax embedding of SQL in PHP looks very simple, yet there are several issues and implicit design decisions in this example, which all contribute to making this embedding actually work as one would expect. In general, defining the syntax of the combination of a host and guest language involves the following decisions: What are the requirements for a suitable *grammar formalism* for expressing the combination of the host and guest languages? How to design the *syntax for quotations*,*.i.e.* the transition from the host language to the guest language? How to design the *syntax for anti-quotations*, *i.e.* the transition from the guest language to the host language. How are guest language constructs bound to names (*e.g.* regular expressions in JavaRegExp) and how can they be *composed* in the host language (*e.g.* dynamically composing SQL queries in PHP). The following subsections discuss these topics. Concerning the grammar formalism, we will finally end up using the SDF syntax definition formalism, yet, we will highlight some pitfalls in this formalism that can inspire the development of improved formalisms for composing language.

3.3 Grammar Formalism

The grammar formalism that is used for defining the syntax embeddings should at least allow modular extension of grammars. This means that the formalism should allow the non-terminals of a syntax definition to be extended in a modular way, *i.e.* the non-terminals need to be *open*. If inline modification of the host language grammar is required to introduce an embedding of a guest language, then readability and maintainability is compromised, just as inline adaptation of software components is in general undesirable. The syntax definition of the host and guest language should be able to evolve independently. Also, inline introduction of a guest language embedding might disable the use of multiple extensions, which we will discuss in Section 4.

Context-free syntax The composition of grammars requires the grammar formalism to use a composable class of context-free grammars. However, only the full class of context-free grammars is closed under composition. All proper subclasses of the context-grammars, for example LL and LALR grammars, are not composable. As a consequence, if the guest and host language are composed in a formalism that is based on a subclass, then the result is not guaranteed to be in the same subclass, which is not allowed since this particular subclass has usually been chosen to enable the use of a particular parsing algorithm. In such a formalism, these composition problems can then only be solved by modification of the grammars, which does not scale to multiple embeddings.

On the one hand, the full class of context-free grammars has the attractive property that syntax definitions can be composed. On the other hand, this class does not guarantee the absence of ambiguities. Indeed, ambiguities arise frequently in syntax embeddings, so design decisions are necessary for dealing with ambiguities. In Section 5 we will discuss the options for resolving ambiguities.

Lexical syntax Similar to the context-free syntax of languages, the lexical syntax need be composed as well. Usually, the lexical syntax of a language is specified as a set of tokens, defined by regular expressions. A scanner splits the input into tokens, which are consumed by the parser. The advantage of this approach is that lexical analysis based on regular expressions is very efficient and reduces the amount of work for the parser. However, if languages are combined, then this approach to lexical analysis gets more problematic. The lexical syntax of the host and guest languages are usually different, *i.e.* they have a different set of tokens. In particular, the set of tokens of a guest language is often not a *subset* of the tokens of the host language. For example, the languages can have a different set of reserved keywords, literals, operators, and layout (whitespace and comments). Hence, combining syntax definitions of different languages introduces different lexical contexts.

A first solution is to ignore this problem. The set of tokens of the host and guest languages could be unified, resulting in a lexical analyzer that recognizes all tokens in every context. This is easy for the implementer of the embedding, yet it is often not user-friendly. For example, reserved keywords in the guest language will become reserved in the host language and vice versa. In some cases this solution is impossible, for example if the token boundaries of the two languages are different. Bali, from the pioneering AHEAD/JTS tool set [3], uses this approach (see [11] for a discussion).

A second solution is to make lexical analysis aware of the context by introducing lexical states. If the transition from the host to the guest language and back can be determined in a lexical analyzer (*e.g.* the guest language is quoted using distinctive tokens), then lexical states can be used to change the lexical context. In particular, this would be possible for many embeddings for code generation, such as Meta-AspectJ [27] and JavaJava [10]. However, for many embeddings lexical states become unmanageable. The transitions between the lexical states are based on a careful analysis of the *complete* language. If new tokens are added by embedding one or more guest languages, then the transitions might no longer be correct. In this case the scanner has to be rewritten, which is not acceptable if we want to realize language libraries.

A third solution is to improve the management of lexical states by controlling the state of the scanner from the parser. This is a major complication of the interface between the scanner and the parser and seems to be rather unpopular in practice.

A fourth solution is to let the scanner produce all possible interpretations of tokens. This is an attractive solution, since it basically has no disadvantages, but it is only possible if the token boundaries of the host and guest languages are exactly the same, which is often not the case. In some cases, a fifth solution can solve the problem of incompatible token boundaries. If the embedded guest language can be scanned with the host scanner, but will produce incorrect tokenizations, then this scanner could still be used. However, the parser can consume the invalid tokens as a plain sequence without any additional structure and delay parsing of these sections. Important for this solution is that the parser can recognize the end of the guest fragment by some unique token that terminates it. The official AspectJ compiler uses this approach for parsing pointcuts [9].

All these solutions have their particular disadvantages, and though they might be suitable for particular applications of syntax embedding, they cannot be used as a solid foundation for realizing language libraries. Instead, declarative specification of the lexical syntax of a language is required, under control of the same module system as the context-free syntax. The SDF syntax definition formalism provides this, by integrating lexical syntax and context-free syntax in the same formalism. SDF is implemented by using scannerless parsing, which means that lexical analysis is omitted and the parser operates directly on the individual characters of a source file. We will discuss the various parsing algorithms and their complications later.

3.4 Grammar Mixins

Figure 4 illustrates that the syntax definitions of the host and guest languages are imported in a new SDF module to create a combination of the languages. However, the plain combination of the two languages is not actually what is needed for embedding SQL in PHP. For example, we

do not want an SQL `Expr` to be combined with an `Expr` from PHP and vice versa. Instead, what is needed is a definition of SQL that can be imported to be placed in an arbitrary context, without automatically interfering with all the non-terminals with the same name.

This is realized by using *grammar mixins* [1]. In the context of object-oriented programming, mixins are abstract subclasses that can be applied to different superclasses (*i.e.* are parameterized in their superclass) and in this way can form a family of related classes [6]. In the context of syntax definitions, grammar mixins are syntax definitions that are parameterized with the context in which they should be used. Only productions for non-terminals that have the same name *and* occur in the same context are composed. Multiple grammar mixins can be subject to mixin composition, which completely composes the languages represented by the grammar mixins.

SDF grammar mixins Grammar mixins can be implemented in several ways and are not in particular bound to the SDF syntax definition formalism. Our implementation of grammar mixins in SDF is based on existing parameterization features of SDF: parameterized modules and parameterized non-terminals. Figure 5 shows the SDF implementation of a grammar mixin module for Java. An SDF grammar mixin is an SDF module that imports another module [11] defining the language to be mixed-in. Furthermore, it has a formal parameter [10] declaring the context in which the mixin is to be placed. In essence, the actual value of the parameter identifies a mixin composition. By convention this parameter is called `Ctx` (for context) and the module name has the suffix `Mix`. In order to make the base grammar 'context-sensitive', the mixin module renames [12] all the non-terminals of the imported syntax definition to parameterized non-terminals, *i.e.* non-terminals parameterized with the context (`Ctx`) in which they can occur. All the non-terminals of the syntax definition have to be renamed, which is typically something that is not desirable to do manually. Therefore, the grammar mixins are actually *generated* by the tool `gen-sdf-mix`, which produces an SDF grammar mixin module for a given syntax definition.

```
module JavaMix[Ctx][10]
imports Java[11]
  [CompilationUnit => CompilationUnit[[Ctx]][12]
   TypeDec         => TypeDec[[Ctx]]
   Expr            => Expr[[Ctx]]
   ... ]
```

Fig. 5. SDF grammar mixin for Java.

A typical syntax embedding based on grammar mixins now imports the grammar mixin modules for the host and the guest language, with some symbol for the context in which they are to be used.

```
module SQL-in-Java
imports SQLMix[SQLCtx] JavaMix[JavaCtx] exports ...
```

The actual value of the formal parameter `Ctx` (in this case `SQLCtx` and `JavaCtx`) is not relevant: any symbol can be chosen, as long as both symbols are different. The value serves as the identification of a mixin composition, but does not have any particular meaning itself.

Reserved keywords In SDF reserved keywords are defined *per non-terminal* using *reject productions* [24], which are productions annotated with the `reject` keyword. Combined with grammar mixins, this feature provides context-specific reserved keywords: keywords are only reserved for the non-terminal in a specific context. Hence, *different* reserved keywords are supported for the contexts of the host and guest languages that have been combined.

Layout Layout (whitespace and comments) should be under control of grammar mixins as well. For example, Java uses `//` as end-of-line comments, but in XPath `//` is part of a path expression. So, layout should just like all language constructs be context-specific and layout of the host and guest languages should not be combined, unless explicitly done by mixin composition. Unfortunately, this is not the case in SDF, where layout is a global non-terminal. This is a design mistake that will hopefully be fixed in the future.

[1] The term grammar mixins has also been proposed by [3], but more as an abstract concept. In fact, this abstract concept is quite related to our concrete grammar mixins

Advanced applications Grammar mixins provide more control over the composition of languages. In the most common scenario of syntax embeddings, the mixins will only be used to give the non-terminals of the involved languages their own context to prevent unintended combinations. So, there are no grammar mixins that are actually composed using mixin composition. However, grammar mixins scale far beyond these simple embeddings. For example, an embedding of a guest language could import a *second* instance of the host language that could be placed in the context of the guest language itself, or as another instance of the host language for specific purposes. These multiple instances of the host language can be used to reserve keywords only in a specific context, for example in anti-quotations or meta-variables. These more advanced features are used in the syntax definition of AspectJ [9], where plain Java is mixed with pointcuts, name patterns, and aspects. These different instances of the Java language can be extended, customized, or restricted separately. For example, context specific reserved keywords can be defined in this way (a design choice of the abc compiler), or Java can be extended only for some specific context, for example to add support for the AspectJ `proceed` method invocation to advice declarations.

3.5 Designing Syntax Embeddings

The most important part of the implementation of a syntax embedding is the design of the syntax for embedding a guest language in the host language. In general, the desired syntax is completely dependent on the specific application domain of the extension, yet for some categories guidelines can be given and the general problems can be listed. Most of the design decisions are concerned with the inherent problem of ambiguities at the boundaries of the host language and the guest language. This is the consequence of using the full class of context-free grammars: its composition properties are most useful, but we have to live with the possibility of ambiguities. Unfortunately, deciding upon the quotation and anti-quotation tokens requires some insight in the syntax of the host and the guest languages. Little support is available for giving advice on these tokens, since analyzing ambiguity of context-free grammars is undecidable and sadly there is not much work available to still have some useful results based on heuristics and approximations.

Quotation and anti-quotation In code and string generation quotations and anti-quotations are often highly ambiguous if the quotations of the various guest non-terminals all use the same syntax. There are three solutions to this problem. First, the number of quotations can be restricted. For example, if the embedding allows the quotation of a list of statements as well as a single one, then a quotation of a single statement will always be ambiguous. In some applications, having only a quotation for the list of statements might be sufficient. Second, the quotations and anti-quotations can be explicitly tagged with their syntactic type, basically introducing different quotation symbols for every quotable non-terminal (see Section 2). This solves the ambiguity problem, but the user now needs to know when and how to use these tags. Furthermore, the tags obfuscate the code and often feel redundant to the user. In some cases explicit tagging can be made less unattractive by using keywords from the guest language. For example, the anti-quotation of an optional where clause of an SQL query can be tagged using `WHERE?`, which looks like the keyword `WHERE` from SQL.

```
"WHERE" "?" "${" E "}" -> Where⟦SQLCtx⟧?
```

Third, the ambiguities can be preserved by parsing the ambiguous source file to a parse forest. The ambiguity can then be dealt with at a later phase (see Section 5), or can even be ignored if the exact representation is irrelevant. For the user this is by far the most attractive solution, since the embedding is not restricted and no knowledge about the ambiguities is required.

In some embeddings it can be attractive to mirror the quotation symbols in the anti-quotation symbols. For example, for an embedding of XML the quotation symbols can be chosen as `%>` and `<%` and the anti-quotation symbols as `<%` and `%>`. In this way, generation of XML with inline code is more natural: `%> <ul> <% ... %> </ul> <%` looks like two separate quotations with some code in between, while it is actually a single quotation with an anti-quotation.

Composing guest language constructs In some applications the guest language constructs do not assimilate to expressions, but rather translate into statements that perform side-effects on objects. An example of this is bytecode emission in the Kawa bytecode library for Java. Anti-quotation is then more problematic, since the anti-quoted code itself is a statement as well, which needs a context to apply the statements to. In this case, new reserved identifiers could be added to the host language (similar to `this`) that are always bound to the right context. This variable could even be added to the context of anti-quotations only by using grammar mixins.

Generic syntax embeddings Syntax embeddings that generalize over multiple host languages can be defined in a generic way by using an SDF module that is parameterized with the expression non-terminal of the host-language. This is illustrated in Figure 6, where the module parameter `E` represents the host expression non-terminal.

```
module Embedded-TinySQL[E]
imports TinySQLMix[SQLCtx]
exports context-free syntax
  "<|" Query[[SQLCtx]]  "|>" -> E
  "${" E "}" -> Expr[[SQLCtx]]
  "$str{" E "}" -> CharString[[SQLCtx]]
```

Fig. 6. Generic Embedding of TinySQL

3.6 Parsing Syntax Embeddings

For realizing language libraries, fully automatic parser generation is very important. Spending development time on a parser for a combination of a guest and host language does not scale to the large number of language libraries (and their combinations) that will be used. However, the parser for syntax embeddings will replace, or at least precede, the existing parser of the host language, for example the parser for Java in a Java compiler. Therefore, the user experience of the generated parser are very important: acceptable performance, decent error recovery, and good error reporting are necessary.

Parsing algorithms As mentioned before, syntax embeddings require the full class of context-free grammars. This reduces the number of parsing algorithms that can be used. Currently, the best studied algorithms for parsing possibly ambiguous context-free grammars are generalized-LR [23, 20] and Earley [13] parsing. Scannerless parsing [21], which is an important feature for syntax embedding, has been integrated with generalized-LR in the implementation of SDF [24]. Recently, Earley has been adapted to do scannerless parsing as well, but no publication or implementation seems to be available for this work. The performance of generalized-LR depends heavily on the grammar: the algorithm has been designed to perform in linear time for deterministic grammars.

Packrat parsing [14] is another candidate for parsing syntax embeddings. Packrat parsers are used to parse languages defined by *parsing expression grammars*, which are closed under composition, intersection and complement. Scannerless packrat parsers are available and have been shown to perform very well [15] compared to current GLR parser implementations. Unfortunately, packrat parsers are not able to produce all possible alternatives for ambiguities: they apply an ordered choice to alternatives. Hence, if ambiguity preservation is required, then packrat parsing is not an option. Also, it is important to pay close attention to the order of alternatives, since packrat parsers are not fully backtracking. Due to memoization, successful choices are committed locally, so they cannot be revisited later.

Error reporting and recovery Scannerless and generalized-LR parsing have attractive properties for parsing syntax embeddings, yet they introduce complications in error reporting and recovery. Due to the forking LR parsers of generalized-LR, it is not immediately obvious how existing techniques for error reporting and recovery of LR parsers can be applied. Scannerless parsing complicates matters even more, since the tokens themselves are now being parsed by the parser as well and error reporting for scanner-based parsers is usually done in terms of tokens. Currently, SGLR reports errors in terms of characters, without any knowledge about related tokens. For example, if a keyword `public` is required, but an identifier `publicx` is found, then the parser

would report an expected character after `public`, which is rather unlikely to be of any help in explaining the error.

Also, declarative mechanisms are needed to specify strategies for error recovery. In practice, grammars are often extended to handle syntactical errors and gracefully continue parsing to report as many errors as possible. If languages are being extended, these error recovery rules might become invalid and might conflict with the language extensions. In generalized-LR parsers this could for example cause inefficiency or ambiguities (if so, then it would have caused shift/reduce conflicts in an LR parser generator).

To make fully automatic scannerless parsing of context-free grammars applicable, research is necessary in this area of error reporting and recovery of generalized parsers.

4 Combining Syntax Embeddings

Having a single extension of a host language available is useful, but many applications require *multiple* extensions to be available in a single source file. For example, in web applications, the main application domain of StringBorg, XML, SQL, Javascript, and Shell, are often used together. As another example, multiple extensions are needed in program transformation systems, where sometimes multiple object languages are manipulated in the same meta program, *e.g.* as source and target language of a transformation.

To truly realize the use of language extensions as language libraries, the user of the extensions should be able to select a number of extensions that must be available to the source code of an application. However, the implementation of extensions as single, closed extensions of the host language will disable the use of *multiple* extensions in the same source file, since the parsing of the syntax extensions cannot be applied in sequence and cannot be composed either if deployed as a complete parser. Hence, extensions should preferably not be deployed as closed extensions of the host language, but rather as separate plugins that can be selected by the user and combined *on the fly* by the system. Also, a system supporting syntax extensions should preferably not be restricted to a fixed number of extensions built into the compiler of the host language. Instead, extensions provided by *third parties* should be supported.

These requirements lead to the need for *independent extensibility* of the host language. For independent extensibility, the language library, which consists of a syntax embedding and an assimilation, needs to be expressed in a truly modular way. Also, the components of language libraries must be *combined efficiently* for a particular selection of a number of (compatible) extensions by the user. In this section, we discuss how this can be achieved for syntax embeddings. Modular syntax definition is a useful basis for this, but more is necessary to realize efficient independent extensibility of syntax. We reflect over the possibility of further optimization by deploying parse table mixins.

Independent extensibility For the embedding of a guest language in a host language, the previous section explained how a new module is introduced that imports the host and guest language, and combines them at some specific places by defining additional productions. Using grammar mixins, the two language are completely separated, except for the places where explicit transitions have been defined. For the combination of multiple extensions, a similar approach can be used. For example, if both SQL and XPath are to be used in Java, then a new module can be defined, which imports both embeddings. Note that we import the Java syntax as a grammar mixin, and use the generic embeddings of SQL and XPath, with `Expr⟦JavaCtx⟧` as the non-terminal of the host language. From this new module a parser can be generated that parses Java programs that use embedded SQL and XPath. Every combination of syntax embeddings requires the definition of such a module, which is clearly not desirable to do by hand. However, a compiler can generate such a module automatically, given the extensions selected by the user.

```
module SQL-XPath-Java
imports
  Embedded-SQL[ Expr⟦JavaCtx⟧ ]
  Embedded-XPath[ Expr⟦JavaCtx⟧ ]
  JavaMix[ JavaCtx ]
```

Yet, this situation is still not optimal, since the parse table is constructed from scratch for every combination of host and guest languages. Parse table generation is rather expensive, in particular for syntax embeddings, which often import multiple language to constitute one big language. Therefore, it would be very useful if the grammar mixins of guest language could be deployed as pre-compiled parse tables, that can be composed efficiently on the fly for a specific combination of language extensions. Some research has been done for incremental parser generation [16, 12], however, this work does not address the modular composition of already generated parsers. We will address this feature in future work.

5 Assimilation

The assimilation phase provides the actual implementation of the embedded syntax of the domain-specific language. In this phase the embedded language constructs are removed from the source program by a translation to the host language, using any necessary translation scheme. The implementation of the assimilation phase largely depends on the application for which the embedded language is intended. The complexity of the implementation depends on the particular combination of the embedded language and the host language. The design of the assimilation is influenced by the requirements to make embeddings a) easy to understand b) composable c) analyzable. In this section we discuss the design and the issues involved in the implementation of a range of assimilations.

Assimilation rules are the basic transformation steps of the assimilation that have a pattern of guest code at their left-hand side and produce a pattern of host code at their right-hand side. In our examples the assimilation rules are implemented in the Stratego program transformation language. Figure 7 illustrates two assimilation rules for the embedding of Java in Java. The first rule [13] assimilates an array type by generating invocations of the Eclipse JDT API for the representation of Java programs in Java. The second rule [14] assimilates a field access by creating a `FieldAccess` object and invoking some methods on it for initialization. The assimilation rules are applied by an *assimilation strategy* [16] that traverses the program and applies the rules where necessary. Most assimilation strategies have the same structure: they traverse the program topdown and apply the assimilation rules if a quotation is found. If an anti-quotation is found [15], then assimilation is stopped and the assimilation strategy is invoked recursively.

```
Assimilate : 13
  ArrayType(type) -> e|[ _ast.newArrayType(e) ]|
  where e := <Assimilate> type

Assimilate : 14
  Field(e,y) -> |[
    {| FieldAccess x = _ast.newFieldAccess() ;
       x.setExpression(e1); x.setName(e2);
     | x |}
  ]|
  where <newname> "expr" => x;
    [e1,e2] := <map(Assimilate)> [e,y]

Assimilate = 15
  ?AntiQuote(<assimilate-strat>)

assimilate-strat = 16
  alltd(?Quote(<Assimilate>))
```

Fig. 7. Assimilation of Java in Java

Program Representation One of the design decisions to make is whether the assimilation should operate on a parse tree or on an abstract syntax tree (AST). An advantage of applying assimilations to ASTs is that they become easier to implement. ASTs are more concise, since irrelevant details, such as whitespace, comments, and keywords have been removed. A disadvantage is that in the back-end of the assimilation a pretty-printer for the host language will be necessary if the assimilation is not integrated in the host language compiler itself. The advantage of applying assimilations to parse trees is that the assimilation can preserve the program layout in this way and that no pretty-printer for the host language is necessary. This is in particular useful for interpreted programming languages, since type errors cannot be prevented by an extended typechecker for the host language (there is no typechecker).

Scope of Assimilation Rules The scope of a transformation [26] indicates the parts of the source and target program that are involved in the transformation. The scope of an assimilation has a major influence on the complexity of an assimilation.

Local-to-local assimilations are the most attractive class of assimilations. In this class the assimilation can be expressed by mapping guest code fragments directly to host code fragments, which are at the exact same place in the abstract syntax tree as the original guest code. Local-to-local rules are easy to implement, since the rules are all independent and do not influence the assimilation strategy. The first example assimilation rule [13] is an example of a local-to-local rule. The second assimilation rule [14] is local-to-local as well, but only due to an extension (expression blocks) of the host language (Java) that reduces the complexity of a more complex local-to-global assimilation to local-to-local.

Local-to-global assimilations are a bit more complex. This class of assimilations produces host code not only locally, but globally as well (*i.e.* higher in the abstract syntax tree). Examples of typical local-to-global assimilations in the context of Java are adding new import declarations, member classes in the enclosing class, new local variables, or even completely new compilation units. In general, the problem that leads to the use of a local-to-global assimilation is that the target of a local-to-local assimilation rule (*e.g.* an expression) does not allow the operations that are required. For example, executing statements, declaring new methods, or introducing import declarations. In this case the assimilation affects the host program at multiple places. Local-to-global assimilations add complexity for several reasons. First, the target of a global assimilation needs to know that code might be generated at this location. Second, the target may actually be quite difficult to find, for example, finding the nearest block-level for generating new statements from the expression-level is not completely trivial.

The assimilation of a field access illustrates a solution for local-to-global assimilations. For the assimilation of a field access a new local variable needs to be declared. This is not possible at the target of assimilation, which is an expression. Since this situation is extremely common we have added a feature to Java, the host language, to make such assimilations local-to-local. Expression blocks (`{| bstm* | e |}`) allow variables be declared and statements to be executed at the level of an expression. In general, local-to-global assimilations are usually implemented using dynamic rules, which allows assimilation effects to be defined locally, yet applied globally. This is illustrated in an assimilation rule for Swul buttons [17] in Figure 8, which locally defines the assimilation effect of generating an import declaration.

```
Assimilate : [17]
  |[ button of c ]| ->
  e |[ {| x= new JButton(); ... |} ]|
  where ... ;
    rules(
      ImportDec :=+ |[ import javax.swing.JButton; ]|
    )
Assimilate = [18]
  ?Quote(<id>, attrs)
  ; {| FactoryName
    : l := get factory name
    ; rules(FactoryName : _ ->  TypeName(Id(l)))
    ; Assimilate
    |}
ClassInitializer : [19]
  |[ static { bstm1* } ]| -> |[ static { bstm2* } ]|
  where {| FieldModifier
        : rules(FieldModifier :+ _ -> |[ static ]|)
        ; <swul-assimilate> bstm1* => bstm2*
        |}
```

Fig. 8. Scope of Assimilation Rules

Global-to-local assimilations are typically assimilations that need context information from the original program for a local assimilation, such as the current package, class, method, or even type information. For global-to-local assimilations dynamic rules are again useful. The assimilation of quotations in StringBorg [18] illustrates the use of a dynamic rule to define the name of a Java factory class that is to be used to construct objects for the code of the quotation.

Global-to-global assimilations need context information from the source program and also produce global host code. An example of this is partially illustrated by the custom traversal for

class initializers [19] in Swul. In this assimilation fields are generated globally from the embedded guest code. If the guest code occurs in a static context, then the generated field has to be static as well. For this reason, field modifiers are collected in the global-to-local direction, which are then used to generate fields using a local-to-global transformation.

Hygiene Assimilation rules frequently introduce new names, for example for local variables and classes. To avoid name capture, the generated names should all be unique. The assimilation rules use the strategies `new` or `newname` to generate a globally unique identifier. This is not enforced by the system (*i.e.* it is possible to use an arbitrary identifier for generated names), but in practice developers always seem to use the correct way to introduce new identifiers.

Genericity The assimilations we have discussed until now are specific for a certain guest language. For example, the assimilation rules for Java in Java are specific for the embedding of Java and the translation to the Eclipse JDT API. In this way meta-programming (*i.e.* an assimilation) is required for every guest language. This is unnecessary if the embedding is actually part of a family of embeddings that all implement a similar assimilation. Indeed, some assimilations are generic in the guest language.

The assimilation of concrete object syntax for Stratego [25] is an example of such a generic assimilation. The representation of object programs in Stratego is directly based on the syntax definition of the object language, thus the transformation from the guest code to Stratego can be implemented generically. StringBorg is an example of a system where the API for the representation of guest code (SQL, XPath, Shell, etc.) is *generated* from the syntax definition, which implies that there is a fixed mapping. Thus, the StringBorg assimilator is completely generic in the guest language. This is a major advantage, since no meta-programming expertise is now necessary to add a new guest language to the system. The assimilation of StringBorg is implemented on parse trees to avoid the need for pretty-printers and preserve the layout of the program. In the case of Stratego, the assimilation is integrated in the compiler and is implemented on abstract syntax trees. In some cases the assimilation can even largely be *host language* independent. To reduce the amount of work for adding a new host language to StringBorg, the assimilation first translates to an abstract representation of host code. Host language specific back-ends implement the mapping to Java, PHP, and Perl. Some assimilations are almost generic but require some additional minor configuration to make them completely generic. For example, the Stratego compiler supports the specification of additional desugaring rules that should be applied to concrete object syntax before assimilation. StringBorg accepts a configuration file, specifying the escaping rules for the guest language, which cannot be derived from the syntax definition itself.

Semantic Analysis and Error Reporting If the host language can be typechecked statically, then it is useful to apply the typechecker before assimilation, to keep the error reports as close to the original source as possible. This is even more important if the assimilation is not local-to-local, since local-to-global assimilations will make it difficult for the user to trace where the assimilated code comes from. To perform typechecking on the extended host language an extensible typechecker is required that can be extended with typing rules for guest languages. For many embeddings these typing rules are very simple and follow directly from the assimilation rules. Besides being able to locate typing errors, an explanation of these errors would be most useful. For this, guest language specific specifications for error reporting could be introduced. We have not experimented yet with such a mechanism.

Assimilation of Ambiguities If ambiguities are preserved by the parser, then the assimilation has to deal with ambiguities. Note that these ambiguities only occur at the boundaries of the guest and host language.

Type-based disambiguation A solution for removing ambiguities is to first assimilate the forest of the extended language to a forest that only contains host language constructs and apply a disambiguating type checker to eliminate the alternatives that violate the typing rules of the host

language [10]. In this approach the granularity of the assimilation rules is very important. Ambiguities can occur in arbitrary locations and if the assimilation rules assimilate guest fragments that are too large (*i.e.* parts can be ambiguous) then the assimilation rules will not be complete.

The big advantage of this disambiguation solution is that it is generic in the guest language: the disambiguating type-checker can be applied to arbitrary embeddings of guest languages, including combinations. The disadvantages of this solution are that it is only applicable to languages with a static type system and manifest typing and that it requires a considerable amount of work to implement a disambiguating type checker. Also, without additional methods for tracing the guest code after the assimilation, the type checker will report type errors in terms of the assimilated code.

Type-based disambiguation can also be applied *before* assimilation. In this case, the type-checker of the host language needs to be extended with typing rules for the guest language. Fortunately, this is desirable anyway for performing semantic analysis of the combined source file. The typing rules can in some cases be automatically derived from the assimilation rules, which avoids duplication. The advantage of this approach is that errors can be reported in terms of the guest language.

Ambiguous target representations The assimilation could preserve the ambiguities as well, by assimilating the guest code to an ambiguous target representation. This solution is particularly attractive for applications of code and string generation, where the exact representation of the code is not relevant, as long as it is composed correctly in one way or another (*i.e.* using one of the alternatives). For transformations and DSL embeddings this solution is in most cases not suitable, since the exact representation of the guest code is often relevant for these applications.

For the preservation of ambiguities at run-time a distinct ambiguity type could be used, however, this might require the user to have knowledge of the possibility of ambiguities. In some languages, this type could be parameterized with the types of the ambiguous alternatives. In this way, the type system of the host language could statically guarantee that the guest code is constructed correctly. Alternatively, the type could be checked at run-time, which is already done in the current, non-ambiguous, embeddings of StringBorg in PHP and Perl. Finally, a single type could be used for all guest code representations. This might increase the usability, but decreases the static guarantees in a statically typed language. Run-time checks can be used to verify that code is composed correctly.

6 Related Work

Syntax Embedding Macro systems usually restrict the syntax that can be introduced. For an extensive review of the relationship to JSE [1], Maya [2], Metafront [5], JTS [3], and camlp4 [22] we refer to [11] and [10]. There are a number of parser generators that could be applied for the parsing of syntax embeddings. Harmonia's Blender [4] and Cardelli's extensible syntax have been discussed in detail in [11]. Packrat parsing [14, 15] has been discussed in Section 3.6. The Polyglot parser generator [19] supports modular adaptation of LR-grammars, which has been discussed in [8].

Furthermore, there is a wide range of applications that apply a form of syntax embedding. In these particular applications (Template-Haskell, MetaOCaml, Meta-AspectJ, SafeGen, etc.) the parsing problem is often reduced because the host language is embedded in itself or development time for parsing this combination of languages is acceptable. Hence, no general method for language libraries is required, though they could profit from such a facility. Recently, support for quoting expressions has been introduced in C#. C# does not introduce a syntax for quotation, but infers quotations from the type of variables and parameters. One of the main purposes of this facility is to use C# expressions to express query expressions, *e.g.* SQL, without introducing syntax (see also [8]).

Assimilation Macro systems usually only allow a straightforward local-to-local mapping from the introduced syntax to the host language. Our assimilation rules and strategies allow local-to-global and global-to-local transformations, which is somewhat related to the desire for macros

to reach out and touch somewhere [17], one of the ideas leading to AspectJ. Macro systems that allow context-free grammars as macro arguments often provide more advanced facilities for transformations. For example, Metafront [5] associates transformations to all productions and additionally guarantees termination of the transformation.

Open Compilers Open compilers such as Polyglot [19] are not designed to facilitate DSL embedding in particular. Instead, they are designed for the introduction of new language features that invade the language, such as new types, optimizations, concurrency features, etc. The requirements for a system that supports the implementation of syntax embedding and assimilation by non-compiler experts are rather different. More experience with extensions is needed to give definite answers on the requirements.

Methodology In [18] the authors state that work is needed on DSL development methodologies, since the use of a DSL can provide major benefits, but the development of DSLs often raises many questions without clear answers or resources on the right decisions to make. Though our work chooses a particular approach to DSL development, our analysis of the design space in this area contributes to the discussion on methodologies. Also, a major focus of our work is to make the implementation of extensions as easy as possible by not imposing technical constraints on the syntax embedding and assimilation.

Domain-Specific Embedded Languages The term domain-specific embedded (or internal) language (DSEL/EDSL) is often used for a DSL defined within the host language, that is, without syntactically extending the host language. DSELs are popular in languages that have a flexible syntax, such as Haskell (user-definable infix operators) and Ruby. In the case of Ruby, there is extensive support for introspection, intercession, code generation, evaluation and bindings, which reduces the need for interpreting the DSL. The heavy use of run-time code generation is largely cultural (since similar functionality is available in languages such as Perl and Python) and due to seemingly irrelevant details, such as multi-line string literals. Considering the syntax, the main disadvantages of embedded domain-specific language approaches is that the syntax cannot be clearly defined separately, but has to be crafted carefully, based on the constraints imposed by the host language.

7 Conclusion

Language libraries provide an approach to bring general-purpose programming and domain-specific programming together. In this paper we have given an overview of the applications of and techniques for realization of language libraries through syntax embedding and assimilation. We have outlined a general architecture for implementation of language libraries and discussed the design choices for instantiating this architecture. Regarding syntax embeddings we discussed constraints on grammar formalisms and parsing algorithms. In particular, we discussed *grammar mixins*, a solution for adapting syntax definitions to specific contexts in a reusable manner. Regarding assimiliation we have discussed scope and genericity of assimilation rules, and the options for the treatment of ambiguities at the interface of host and guest language. We have also identified a number of challenges for further research. An important goal for reaching the full potential of language libraries is the independent extensibility of host languages in order to provide language libraries as independent plugins. That is, language libraries should be deployed seperately and integrated into the programming environment by the end-programmer without recourse to meta-programming. This requires independent extensibility of parsers (or at least parse tables), and extensibility of assimilations.

References

1. J. Bachrach and K. Playford. The Java syntactic extender (JSE). In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'01)*, pages 31–42. ACM Press, 2001.
2. J. Baker and W. Hsieh. Maya: multiple-dispatch syntax extension in java. In *Programming Language Design and Implementation (PLDI '02)*, pages 270–281. ACM Press, 2002.

3. D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: tools for implementing domain-specific languages. In *International Conference on Software Reuse (ICSR'98)*, pages 143–153. IEEE, 1998.
4. A. Begel and S. L. Graham. Language analysis and tools for input stream ambiguities. In *Language Descriptions, Tools and Applications (LDTA'04)*, ENTCS. Elsevier, April 2004.
5. C. Brabrand, M. Vanggaard, and M. I. Schwartzbach. The metafront system: Extensible parsing and transformation. In *Language Descriptions, Tools and Applications (LDTA'03)*. ACM, April 2003.
6. G. Bracha and W. Cook. Mixin-based inheritance. In *OOPSLA/ECOOP'90*, pages 303–311. ACM Press, 1990.
7. M. Bravenboer, R. de Groot, and E. Visser. MetaBorg in action: Examples of domain-specific language embedding and assimilation using Stratego/XT. In R. Lämmel and J. Saraiva, editors, *Proc. of the Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'05)*, LNCS, Braga, Portugal, 2006. Springer Verlag.
8. M. Bravenboer, E. Dolstra, and E. Visser. Preventing injection attacks with syntax embeddings. a host and guest language independent approach. In J. Lawall, editor, *Generative Programming and Component Engineering (GPCE'07)*. ACM SIGPLAN, October 2007. (To appear).
9. M. Bravenboer, E. Tanter, and E. Visser. Declarative, formal, and extensible syntax definition for AspectJ. A case for scannerless generalized-LR parsing. In W. R. Cook, editor, *Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'06)*. ACM Press, October 2006.
10. M. Bravenboer, R. Vermaas, J. J. Vinju, and E. Visser. Generalized type-based disambiguation of meta programs with concrete object syntax. In R. Glück and M. Lowry, editors, *Generative Programming and Component Engineering (GPCE'05)*, volume 3676 of *LNCS*, pages 157–172, Tallinn, Estonia, Sept./Oct. 2005. Springer-Verlag.
11. M. Bravenboer and E. Visser. Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. In D. C. Schmidt, editor, *Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'04)*, pages 365–383. ACM Press, October 2004.
12. L. Cardelli, F. Matthes, and M. Abadi. Extensible syntax with lexical scoping. SRC Research Report 121, Digital Equipment Corporation, Systems Research Center, February 1994.
13. J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.
14. B. Ford. Packrat parsing: simple, powerful, lazy, linear time, functional pearl. In *International Conference on Functional Programming (ICFP '02)*, pages 36–47. ACM Press, 2002.
15. R. Grimm. Better extensibility through modular syntax. In W. R. Cook, editor, *Programming Language Design and Implementation (PLDI'06)*. ACM Press, June 2006.
16. J. Heering, P. Klint, and J. Rekers. Incremental generation of parsers. *IEEE Trans. on Software Engineering*, 16(12):1344–1351, 1990.
17. G. Kiczales, J. Lamping, L. H. R. Jr., and E. Ruf. Macros that reach out and touch somewhere. Technical report, Xerox Corporation, December 1991.
18. M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, 2005.
19. N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for Java. In *International Conference on Compiler Construction (CC'03)*, volume 2622 of *LNCS*, pages 138–152. Springer Verlag, April 2003.
20. J. Rekers. *Parser Generation for Interactive Environments*. PhD thesis, University of Amsterdam, 1992.
21. D. J. Salomon and G. V. Cormack. Scannerless NSLR(1) parsing of programming languages. *ACM SIGPLAN Notices*, 24(7):170–178, 1989. *PLDI'89*.
22. D. de Rauglaudre. Camlp4 reference manual, September 2003.
23. M. Tomita. *Efficient Parsing for Natural Languages. A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, 1985.
24. E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.
25. E. Visser. Meta-programming with concrete object syntax. In D. Batory, C. Consel, and W. Taha, editors, *Generative Programming and Component Engineering (GPCE'02)*, volume 2487 of *LNCS*, pages 299–315. Springer-Verlag, October 2002.
26. J. van Wijngaarden and E. Visser. Program transformation mechanics. a classification of mechanisms for program transformation with a survey of existing transformation systems. Technical Report UU-CS-2003-048, Institute of Information and Computing Sciences, Utrecht University., May 2003.
27. D. Zook, S. S. Huang, and Y. Smaragdakis. Generating AspectJ programs with Meta-AspectJ. In G. Karsai and E. Visser, editors, *Generative Programming and Component Engineering (GPCE'04)*, volume 3286 of *LNCS*, pages 1–19. Springer, October 2004.