Science of Computer Programming 78 (2013) 615–632

Contents lists available at SciVerse ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Incremental concrete syntax for embedded languages with support for separate compilation

Tom Dinkelaker [a,*], Michael Eichberg [b], Mira Mezini [b]

[a] Ericsson R&D, Frankfurt, Germany

[b] Technische Universität Darmstadt, Germany

ARTICLE INFO

Article history:
Received 7 August 2011
Received in revised form 28 April 2012
Accepted 10 October 2012
Available online 6 December 2012

Keywords:
Language embeddings
Domain-specific languages
Language design and implementation
Program transformation
Generic pre-processor

ABSTRACT

Embedded domain-specific languages (EDSLs) are known to improve the productivity of developers. However, for many domains no DSL implementation is available and two important reasons for this are: First, the effort to implement EDSLs that provide the domain's established syntax (called *concrete syntax*) is very high. Second, the EDSL and its underlying *general-purpose programming language* (GPL) are typically tightly integrated. This hampers reusability across different GPLs. Besides these implementation issues, the productivity gains of using EDSLs are also limited by the lack of explicit tool support for EDSL users—such as syntax highlighting or code analyses.

In this paper, we present an approach that significantly reduces the necessary effort to implement embedded DSLs with concrete syntax. The idea is to use *island grammars* to specify the EDSL's concrete syntax. This enables the developer to implement the embedded DSL as a library and to incrementally specify the concrete syntax using meta-data. Only those parts of the EDSL's grammar need to be specified that deviate from the grammar of the GPL. By analyzing an EDSL's implementation using *reflection*, it is possible to provide tool support for EDSLs without having the developer implement it explicitly, such as syntax highlighting. An evaluation demonstrates the feasibility of our approach by embedding a real-world DSL into a GPL.

1. Introduction

Domain-specific languages (DSLs) [1] are frequently employed to foster the productivity of *end users*. To achieve this goal, the syntax and semantics of a DSL are defined such that they resemble a domain's commonly used language. Traditionally, DSLs were used to write self-contained programs; for example, to extract data from a database using SQL. Lately, however, DSLs are more and more frequently embedded into general-purpose programming languages (GPLs) to enable expressing domain-specific problems using the domain's natural concepts and language.

When embedding DSLs, the main problems are the integration and interaction of DSL code with existing code and how to enable the DSL to use the domain's established syntax and semantics. We can distinguish two main approaches w.r.t. how a DSL is embedded in a GPL [2]: *heterogeneous embedded DSLs* and *homogeneous embedded DSLs*. In case of *heterogeneous embedded DSLs*, a specialized tool—in general some kind of *pre-processor*—has to first process the code. The task of the pre-processor is to rewrite DSL expressions into executable GPL code. After that, the compiler of the GPL compiles the code as usual. An example of a *heterogeneous embedded DSL* is *SQLJ* [3] which enables the embedding of SQL code into Java code.

* Corresponding author.
E-mail addresses: tom.dinkelaker@ericsson.com (T. Dinkelaker), eichberg@cs.tu-darmstadt.de (M. Eichberg), mezini@cs.tu-darmstadt.de (M. Mezini).

0167-6423/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2012.12.002

Homogeneous embedded DSLs are "just" libraries developed in the GPL in which they are embedded and that provide domain-specific constructs [4]. Independent of the chosen embedding approach, a GPL is generally called a *host language* if other languages are embedded in it.

In the following, we will describe the tradeoffs of both approaches. The main advantage of heterogeneous embedded DSLs is that they support the domain's established syntax (called *concrete syntax* in the following). This is particularly important as a comparative study [5] has revealed that missing support for concrete syntax significantly lowers the end user's productivity in writing DSL code. The main obstacles of heterogeneous embedded DSLs are threefold. First, to support another host-language, the pre-processor typically has to be reimplemented from scratch. Second, pre-processors are known to be hard to compose, because most of them assume a fixed input syntax. This makes it practically impossible to use multiple embedded DSLs in parallel. Third, the implementation of a DSL requires a formalization of the syntax of the domain-specific language. Additionally, those expressions of the host language need to be specified that are at the syntactic boundary; i.e., where expressions of the EDSL can be used as sub-expressions of the host language. This is also called *syntactic assimilation* [6] and typically requires significant effort.

The problems of heterogeneous embedded DSLs are partially solved by homogeneous embedded DSLs since they are developed in the host language. In the latter case the EDSL's implementation is just a library. Hence, a specification of the syntax and corresponding tools is not required. This helps to significantly reduce the necessary effort. Additionally, using multiple DSLs in the same program is as simple as using multiple libraries. But, by having to reuse the syntax of the host language, the DSL's concrete syntax is typically not close to the concrete syntax of the domain [6,2]. This severely hampers the comprehensibility of such DSLs.

To provide concrete syntax for DSL programs—without suffering from some of the main problems of traditional heterogeneous embedded DSLs—several approaches were proposed that use *meta-programming* [6,7,2,8,9]. These approaches generally suffer from the following issues: First, language developers still have to provide the complete formal syntax for each embedded DSL [6,2,9]. Second, for each supported host language the complete grammar has to be available [6,7] in the format of the approach, which is generally not the case. Third, most approaches [2,8,9] support only one particular host language, which hinders the reusability of EDSL implementations. Overall, the effort necessary to develop an EDSL using these approaches is significantly higher than the effort to develop a homogeneous embedded DSL.

In the following, we propose an approach to implement embedded DSLs that combines the advantages of heterogeneous embedded DSLs with the advantages of homogeneous embedded DSLs and which avoids the respective disadvantages. That is, our approach facilitates the development of EDSLs that provide concrete syntax, but avoids the need to specify the complete grammar of the EDSL and/or the host language. Using our approach the effort to implement an EDSL with concrete syntax is practically identical to the effort necessary to develop a homogeneous embedded DSL for the respective domain. Furthermore, our approach is host-language independent and provides a generic tool that can be used to embed DSLs in various host languages.

The fundamental idea of our approach is that a language developer starts with the implementation of a homogeneous embedded DSL and then—step by step—adds meta-information regarding the concrete syntax of the domain to the DSL implementation. By using so-called *island grammars* [10] the developer only needs to specify those parts of the grammar of the host language that are relevant w.r.t. the EDSL implementation. Furthermore, only those parts of the EDSL's syntax need to be specified that are not compatible with the host language's syntax.

A second major issue with (homogeneous) EDSLs is that they are not explicitly supported by IDEs. This lack of explicit support decreases the end users' productivity [1,5]. In general, IDEs use the host language's rules for syntax highlighting and code completion also for the EDSL code. But these rules often do not fit the domain's concepts well. For example, the keywords of the domain-specific language may be implemented using types, methods and variables of the host language and are correspondingly highlighted. However, from the point of view of the domain-specific language, they should all be highlighted as keywords of the domain. Hence, in such cases the host-language-based syntax highlighting of the EDSL code can even hamper its comprehension. Furthermore, compile-time error messages are also unrelated to the domain and its concepts. As a first step towards solving these issues, we propose an approach that, based on the analysis of an EDSL's implementation, performs domain-specific syntax highlighting for the respective code.

The contributions of this paper are as follows: (1) A comprehensive discussion of the use of island grammars for EDSL specifications. (2) An evaluation that shows that island grammars help to significantly reduce the necessary effort when developing new EDSLs. (3) A concrete approach for the definition of an EDSL's syntax and semantics that facilitates evolution. (4) A proposal for how to incrementally add concrete syntax to an EDSL. (5) The first implementation of a language workbench that supports the usage of multiple EDSLs that are deployed in binary form and which do not need to be recompiled by users. (6) A prototype, which is a set of Eclipse plug-ins that can be downloaded from http://github.com/stg-tud/tigerseyeplugin. It implements the specification of new languages, supports syntax-highlighting for arbitrary EDSLs and can execute those EDSLs.

The remainder of this paper is organized as follows: Section 2 briefly presents necessary background information regarding parsing of programs. Section 3 discusses the development and usage of independent EDSLs. Section 4 discusses IDE support for EDSLs. Section 5 discusses the concept and Section 6 the implementation of our prototype. Section 7 presents the evaluation. Section 8 compares our approach to related work. Finally, Section 9 concludes the paper.

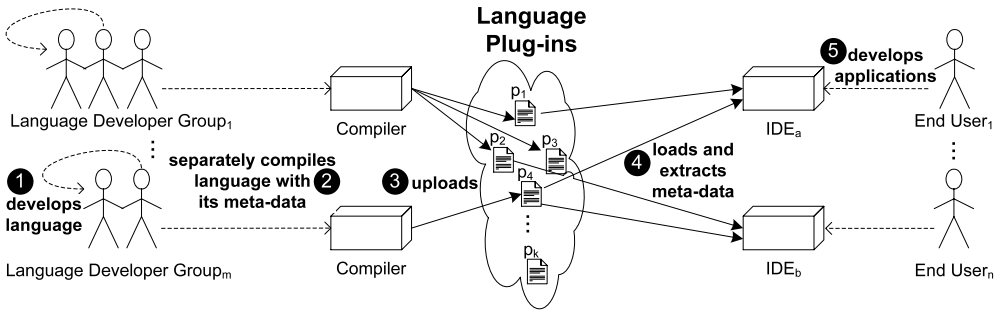

Fig. 1. The relation between developers and users of EDSLs.

2. Parsing programs

Traditionally, to implement a new DSL, first the grammar is specified w.r.t. the domain's established notation using formalisms, such as BNF or SDF [11]. Given the formal definition of the grammar, the developer can then use a parser generator to generate various tools that support processing programs written in the language. For example, often the *lexer*, the *parser*, and the classes to represent the *abstract syntax tree* (AST) are generated.

However, most established parsers only support a specific class of *context-free grammars* (CFGs) [12], such as LALR, LL(k), LR(k) grammars. That is, to be processable by the parser–generator the grammar not only has to be context-free, but it has to have additional properties. For example, it has to be free of ambiguities, it has to be in *Chomsky normal form* (CNF), or it must not be left-recursive. Furthermore, the composition of two or more grammars is generally not supported, because the combined grammar is not guaranteed to have the required properties [6,2]. If we want to support programs that use multiple DSLs, the value of such parser–generators is significantly limited.

Expression grammars [13] is a relatively new approach that enables composition of languages. But, it also does not support all context-free languages, which is generally required to support the definition of "arbitrary" context-free DSLs [6].

Parsing algorithms that support arbitrary CFGs are the SGLR [14], GLL [15], and Earley [16] algorithm. They support the composition of languages, because the combination of two CFGs is again a CFG. In particular, these algorithms support ambiguous grammars by creating a *forest of ASTs* that records all possible ASTs containing all ambiguities. *Disambiguation filters* [17] can then be used to select the right AST from this forest of ASTs.

Island grammars [10] support parsing programs even if only a subset of the productions of the complete formal syntax of the language is specified. In general, an island grammar only completely formalizes the language constructs of interest (the islands). To match the remainder (the water), liberal productions are specified. These liberal productions, called *water rules*, recognize characters from expression types that are not recognized by the islands. Hence, water rules are used to specify the parts of a program that need not be analyzed any further. When composing island grammars of multiple languages, programs can be parsed that mix expressions from those languages in unexpected ways [18]. We propose to use island grammars to avoid what language developers always have to formalize, the complete grammar of the embedded DSL and the host languages. Compared to the traditional use of island grammars as a means to avoid the formalization of a legacy language in the reverse-engineering context, we employ them to reduce the development costs for a new language.

3. Independent development and deployment of EDSLs

As shown in Fig. 1, in our approach the developers of embedded domain-specific languages can independently and incrementally develop (1), compile (2), and distribute (3) the language implementations. Hence, our approach enables division of labor for language development. Given the implementations of multiple EDSLs, users of the EDSLs can then selectively deploy (4) those languages that they require to develop the application at hand (5). Compared to existing EDSL approaches with support for concrete syntax, our approach does not require that language developers need to develop and compile all EDSLs in one step in which the source code of every language implementation must be available.

Language development. As in case of homogeneous EDSLs, our approach enables language developers to independently develop languages in the form of libraries in an object-oriented language. Unlike in case of heterogeneous EDSLs, language developers can still use the host language's concepts for the modular development of a language implementation, i.e., they implement a new EDSL using standard interfaces and classes. In a second step, it is then possible to extend or refine the language by means of *grammar inheritance* [6,19,20]. That is, it is possible to define new syntax and semantics by inheriting from the classes of the existing implementation. To support concrete syntax for the extended or refined language language, developers can incrementally add the necessary meta-data.

Language compilation. In our approach, language developers are still able to compile each of their languages independently from each other. At the end, for each language, a plug-in is built that contains the compiled classes of the implementation. Unlike with other EDSL approaches, a language plug-in just consists of binary code (3).

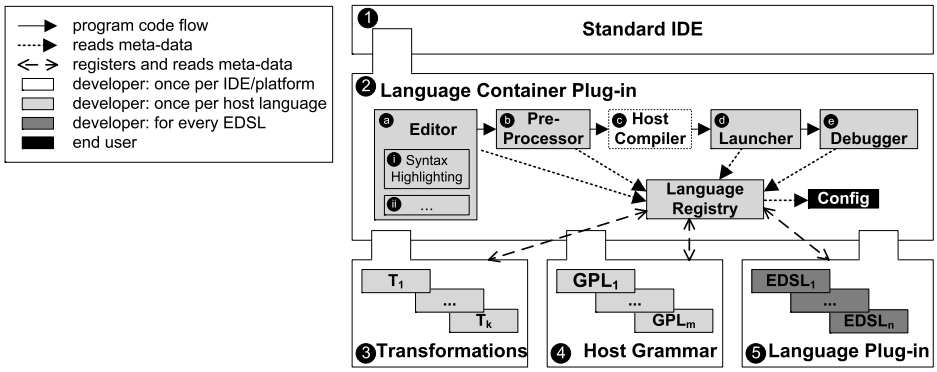

Fig. 2. The language container architecture.

Language deployment. Deployment of new languages is greatly facilitated by supporting the download of EDSLs directly from the Internet (4). The download is triggered when the end user registers a new EDSL. After the download has finished, the container's services automatically adapt themselves to support the EDSL. All information that an end user needs to provide is:

(a) the URL from which the container can download the plug-in binaries,
(b) a unique key for the language, which will be part of the file name extension for stand-alone EDSL program files,
(c) optionally, the preferred keyword color for syntax highlighting, and
(d) if necessary, a selection of options that trigger special transformations for programs of that language.

Unlike other approaches that generate IDEs for (E)DSLs, our language container does not need to be regenerated or recompiled, e.g., when new languages are selected by the end user or updates become available.

Language composition. When combining two languages, we have to compose the languages at the syntax level and the semantic level. To facilitate the composition we rely on important results from related work as explained in the following.

We use an Earley parser [16] which supports composable island grammars [10]. The chosen parser is able to handle syntax ambiguities by creating forests of ASTs. To facilitate the selection of the correct AST, EDSL developers and users can use disambiguation filters [17]. As usual for homogeneous embedding approaches [21], semantic composition takes place at the host-language level, after the domain objects are mapped to objects in the host language. Conflicts—if any—are then identified by the host language compiler. Using this proposed approach makes—in many cases—the composition of EDSLs possible without requiring that an EDSL has to be changed or adapted to enable composition.

Application development. Using our reflective IDE, the end user can write stand-alone EDSL programs or embed DSL statements into programs of the host language. When end users use the primitives of various EDSLs inside their applications, our plug-in takes care of orchestrating the transformation of the application code with embedded DSL statements into an executable form. As long as there are no conflicts[1] this transformation process is completely transparent to the user. That is, the end user can conveniently compile such EDSL programs and run them from the IDE. In general, to help prevent conflicts, quotations such as `#sql(...)` could be used. However, if the EDSL developers have not foreseen possible conflicts, users can always extend the EDSLs to enable conflict-free compositions.

4. The IDE as a language container

As already stated in the introduction, explicit IDE support for embedded languages is crucial, but developers of EDSLs generally do not implement special tool support due to the required effort. In this section, we now discuss how to provide IDE support which keeps the effort required by the EDSL developers at a minimal level. Basically, we propose to extend the host language's IDE with special support to reflect on EDSLs.

Fig. 2 shows the proposed architecture for IDEs that act as language containers for embedded DSLs. For a *standard IDE*, such as Eclipse, (index 1), a *language container plug-in* (index 2) needs to be implemented that adds the generic support required for tool support for EDSLs. The language container has a *language registry* which is used by the services, such as the syntax highlighter, to retrieve an EDSL's meta-data. The *configuration* of the container contains all relevant information regarding user-specific options and the selected EDSLs. The language container itself provides extensions points for *language transformers* (index 3) and for different *host languages* (index 4). Finally, there is a set of *language plug-ins* for EDSLs (index 5) that provide the meta-data for the EDSLs.

[1] In Section 7.3 we will discuss a concrete example.

For every registered EDSL several services are available: (a) services related to editing the source code like *syntax highlighting* (index i) and *code completion*, (b) *pre-processors* that make EDSL programs compilable, (c) *compilers* to translate EDSL programs into a binary form, (d) *launchers* to start EDSL programs, and (e) generic *debugging* support for EDSL programs. These container services are *generic* in the sense that they are not specific for a particular EDSL. These services all rely on the meta-data provided by the EDSLs, which is stored in the registry, and the general configuration information. Using this information the services are specialized for the particular EDSLs. For example, to enable domain-specific syntax highlighting for SQL, the editor services look up the SQL grammar from the meta-data available in the SQL language plug-in, and then use this meta-data to configure the generic syntax highlighter. This architecture and the fact that all services are implemented against the meta-data provided by EDSLs frees the developers of EDSLs from implementing special support for the different services offered by IDEs.

5. Concrete syntax for homogeneous embedded DSLs

In this section, we first elaborate on the implementation of homogeneous embedded DSLs from the point of view of the language developer. After that, we will show how to provide concrete syntax for an EDSL. Finally, the inner workings of our approach are presented before we conclude the presentation of our approach by discussing how to support further host languages.

The running example to demonstrate our approach is the embedding of a subset of SQL in the Groovy GPL, called TinySQL. The goal is to make it possible to write programs like the one in Listing 1. The select statement (Line 2) is defined using TinySQL while the rest of the program is written in Groovy; both languages are seamlessly integrated. The program uses TinySQL's concrete syntax (cf. Listing 2) and prints out a list of peoples' names and ages from the result of an embedded TinySQL statement.

```
1 println "Registered people:"
2 List<Map> result = SELECT Name,Age FROM People
3 result.each{ Map row ->
4   println "Name: ${row.Name} Age:${row.Age}"; }
```

Listing 1. Embedded TinySQL program in Groovy.

```
1 Statement → Query
2 Query → "SELECT" ␣ Columns ␣ "FROM" ␣ Tables
3 Columns → Id | Id ␣ "," ␣ Columns
4 Tables → Id ␣ "," ␣ Tables | Tables → Id
```

Listing 2. The Grammar of TinySQL.

We chose Groovy as the host language for our examples because it has a very flexible syntax. This reduces the effort necessary to develop an EDSLs grammar as we will discuss in the following. However, this is not a crucial/necessary feature; just a beneficial one. Other modern programming languages such as Scala could also be used.

5.1. Defining a homogeneous EDSL

A new homogeneous EDSL is implemented by a set of classes in the host language. These classes have to adhere to the conventions listed below to enable the integration with our approach.

To embed a DSL the following artifacts have to be created:

1. The *language interface* that declares the language constructs of the DSL and which must extend the DSL *marker interface*.[2] In the language interface, the developer declares a method for each expression type that takes sub-expressions as arguments and that will return a computation for a corresponding expression.
2. The *language implementation* defines the execution semantics of the DSL. It consists of a class that must subclass the `Interpreter` class, which acts as a *façade* to a set of classes or third-party libraries. The language implementation class implements the language interface. By implementing each declared method, the class provides execution semantics for each expression type.

In the following, we only discuss the embedding of languages into Groovy, but our approach would be similar for other host languages.

[2] This means the DSL interface does not declare any methods.

Listing 3 shows the interface for the homogeneous EDSL TinySQL. The interface only defines a single method[3] to execute select statements.

```
1 interface ITinySQL extends DSL {
2   List<Map> selectFrom(String[] columns, String[] tables);
3 }
```

Listing 3. The interface of TinySQL.

Given the constraints of the host language, the syntax of the homogeneous EDSL cannot be optimal in most cases. In the following, we use the term *compromised syntax* to refer to the syntax that is provided by the homogeneous EDSL.

Defining the semantics. To implement the execution semantics of an EDSL, a language developer basically just implements the EDSL's language interface. Since the focus of this paper is to discuss support for concrete syntax, we only sketch the implementation of the execution semantics for EDSLs. For further details we refer to Dinkelaker et al. [22].

Listing 4 presents an excerpt of the class `TinySQLInterpreter` that implements the semantics of TinySQL. In general, an EDSL class extends the `Interpreter` class and implements the EDSL's language interface. The `Interpreter` class primarily declares a method (`eval`) to evaluate EDSL programs. For TinySQL, the class implements the language interface `ITinySQL`. Additionally, it has an attribute to hold an SQL connection and implements the `selectFrom` method. The method first checks that all arguments are correct. Then it constructs an SQL statement and uses *Groovy SQL*[4] to execute the statement using a JDBC connection. The method `rows` in Line 9 finally returns a result set as a `List` that contains `Map` objects as rows.

```
1  class TinySQLInterpreter extends Interpreter
2                               implements ITinySQL {
3    private groovy.sql.Sql connection = newInstance("jdbc:...");
4     ...
5    List<Map> selectForm(String[] columns, String[] tables) {
6       ...  //check arguments to be correct
7      String query = "SELECT " + Util.implode(",", columns);
8     query += " FROM " + Util.implode(",", tables);
9     return sqlConnection.rows(query);
10   }
11 }
```

Listing 4. The implementation of TinySQL.

Embedding DSL expressions into a host program. To homogeneously embed DSL expressions into host language programs, the compromised DSL syntax has to be enabled in the host language. Depending on the host language, different mechanisms – e.g., *static imports* in case of Java and Scala, or *closures* and *meta-objects* in case of Groovy – can be used. We call the additional program logic that is necessary to execute a DSL statement the *bootstrap logic*.

For example, to execute SQL statements written using the homogeneous embedded DSL TinySQL (`selectFrom` in Listing 5 Line 5), we need an instance of the corresponding language interpreter (`TinySQLInterpreter`) and have to pass the DSL statement to it (cf. Listing 5). To make the latter possible, we have to first wrap the DSL statement into a closure (Lines 2–4) and have to pass the closure to the interpreter (Line 5). At runtime, when the closure—encapsulating our EDSL statement—is passed to the interpreter, the closure is immediately evaluated and the result is returned.

```
1 println "Registered people:"
2 Closure dslProgram = {
3   return selectForm(["Name","Age"],["Person"]);
4 }
5 List<Map> result = new TinySQLInterpreter().eval(dslProgram);
6 result.each{ Map row ->
7   println "Name: ${row.Name} Age:${row.Age}"; }
```

Listing 5. Program using the homogeneous embedded DSL TinySQL.

[3] This method is expected to return a computation (or value) for the corresponding SELECT expression type. For example, we represent query results with a `List` of `Map` objects. Each row is one of the list items containing a `Map` that holds a key-value pair for each cell, in which the key is the cell's column name associated to the cell's value.

[4] Groovy SQL: http://docs.codehaus.org/display/GROOVY/Tutorial+6+-+Groovy+SQL.

The bootstrap logic, i.e. the closure that wraps the DSL statement and the instantiation of the DSL interpreter, is always implemented in the same way and is automatically generated by our approach, as we will describe later on.

5.2. Enabling concrete syntax for EDSLs

In this section, we discuss how language developers can add support for concrete syntax for their homogeneous embedded DSLs.

Incremental definition of concrete syntax. To incrementally define concrete syntax for embedded DSLs, the language developer adds annotations to their EDSL implementation classes to specify the EDSL's grammar.

For every method that defines an expression type, the EDSL developer adds an annotation `@DSLMethod` to define the concrete syntax for the corresponding expression type. The annotation has the element `production` that defines the right-hand side of the production rule for this method. The production rule is obtained as a string and defines the categories in the production rule.

From each production's string, the necessary meta-data is extracted. In the string, the categories are delimited by the *whitespace escape character*, which is the underscore. One underscore denotes a required whitespace (`_`), and two underscores denote an optional whitespace (`__`). Terminal categories simply use their ASCII representation. Non-terminal categories are defined by argument placeholders, which are prefixed by the *parameter escape character* (by default, `"p"`) followed by the index of the argument in which the non-terminal will be passed to the method. As a result, for each `@DSLMethod` annotation, the EDSL grammar automatically yields a production.

```
1 interface ITinySQL extends DSL {
2   @DSLMethod(production="SELECT__p0__FROM__p1")
3   List<Map> selectFrom(String[] columns,
4                        String[] tables);
5 }
```

Listing 6. The interface of TinySQL.

For example, Listing 6 shows the definition of the TinySQL interface that defines concrete syntax. The annotation in Line 2 defines a production (*Statement* → `"SELECT"` `__` *String*[] `__` `"FROM"` `__` *String*[]) with four relevant categories on the right-hand side: `"SELECT"`, *String*[], `"FROM"`, and *String*[]. By default,[5] the left-hand side category of the production is *Statement* which is expected to be a category of the host language and which identifies where the DSL's expression can be embedded (cf. Line 3 in Listing 9).

The textual encoding of production rules used by our approach resembles familiar notations to ensure that developers can easily understand them. Hence, most OO developers should be able to immediately start developing EDSLs. This addresses the experience of seasoned researchers [23] that many language developers do not want to adopt new notations for defining DSLs. Moreover, using annotations for the specification of the productions saves effort, because the syntax and semantics of an EDSL is defined in one artifact and not spread across different artifacts that use different languages.

Quotation. Given that our approach builds upon homogeneous EDSLs, the user generally does not have to quote DSL expressions. The fact that the EDSL user can omit quotations is an important advantage of homogeneous embeddings. Typically, when integrating concrete syntax of heterogeneous EDSLs into the host, quotation is used in order to delimit expressions of EDSLs and host. Quotation makes it simpler for a combined parser to recognize from which of these languages an expression originates. While quotation makes the task of the parser easier, quotes imply additional work for the user and even may confuse reading and understanding the source code.

In our approach, the developer can voluntarily define a quotation expression as part of an EDSL, because—even for heterogeneous EDSLs—expressions are often well-defined in a certain context. Therefore, the decision if an EDSL should use quotations for embedding is left to the language developer, as quotation is often not necessary.

However, there are cases where a language developer may require to control the integration between an EDSL and a host language. One common reason is to precisely define the scope of an EDSL expression. For example, in Listing 7, in Line 2, the EDSL expression is quoted using `#sql{...}`.[6]

In our approach, quotations are only necessary in case of EDSL keywords that could be misinterpreted or in case a language developer wants to restrict the scope of identifiers. Quotations are directly supported by our approach and do not require special support. It is sufficient to define another `@DSLMethod` annotation that defines a production regarding the quotation (cf. Listing 8 Line 3).

[5] Developers can turn off using embedded expressions at the top level (i.e. *Statement*) of the host grammar by setting the element `topLevel=false`.

[6] `#sql{...}` is not a valid statement in the host language. When a language end user uses a quotation, the end user explicitly escapes from the host language and opens up a scope in which SQL expressions can be used.

```
println "Registered people:"
Map[] result = #sql{SELECT Name,Age FROM People}
result.each{ Map row ->
  println "Name: ${row.Name} Age:${row.Age}"; }
```

Listing 7. An Embedded SQL program in Groovy.

```
interface ITinySQLWithQuotation extends ITinySQL {

  @DSLMethod(production="#sql{__p0__}")
  Map[] quoteSqlStmt(List<Map> map);
}
```

Listing 8. The interface of TinySQL with Quotation.

As it is common for homogeneous embedded DSLs, the types used by the EDSL and the types of the host language are often directly compatible. For example, in Listing 5 the type of the DSL expression has the type as expected by the host language program. However, in general, it is not safe to assume that the types of the EDSL and the types used by the host language program or other EDSLs are compatible. In this case, it is necessary to extend the interpreter (cf. Listing 8) and to implement the method declared in Line 4 to perform the type conversion of the value. As in the previous case, we could directly reuse the standard functionality of our approach and no special support is needed.

Configuring the embedding. The presented `@DSLMethod` annotation offers an efficient mechanism to define concrete syntax for EDSLs that satisfies many needs. But, to cope with special requirements, advanced configuration mechanisms are provided in the form of several optional annotation parameters that enable developers to adjust the default conventions to their needs. For example, the parameter `waterSupported=false` disables island grammars while parsing an EDSL statement. This deactivates tolerant parsing, which restricts our approach to only accept complete grammars. The parameter `whitespaceEscape="  "` changes the whitespace character used in `production` parameters from the underscore to the empty space. This allows the developer to define the concrete syntax without underscores, e.g. `@DSLMethod(production="SELECT p0 FROM p1")`, which may further improve the comprehension of the concrete syntax defined in a `production` element. The element `stringQuotation="'"` changes the *string quotation character*, which by default is "".

5.3. Compiling DSL programs

Fig. 3 gives an overview of the compilation process that transforms programs in concrete syntax into host syntax. This process is explained in the subsequent paragraphs.

Starting the compilation process. The compilation process starts (Step 1), when the DSL end user saves a file in the IDE that contains expressions in concrete DSL syntax. To specify the EDSLs used in a source file users can use a file extension that ends with `dsl`. E.g., `file1.sql.dsl` uses `sql` to specify that the TinySQL DSL is used. Alternatively, users can use the annotation `@EDSL` to specify the used languages. The annotation's elements determine the concrete EDSLs. For example, the annotation `@EDSL(sql,expr)` specifies that the `sql` and `expr` DSLs are used.

Extracting grammars using reflection. For every EDSL the pre-processor looks up the EDSL interface (Step 2a) from the configuration. Next (Step 2b), the pre-processor uses *introspection* [24] to look up all declared methods in the EDSL's interface. From the interface annotation, the pre-processor looks up the host language of the EDSL and loads the host grammar (Step 2c). From meta-data provided in the method signatures and their annotations, the pre-processor extracts the EDSL grammar (Step 2d).

Creating a combined grammar. Next (Step 3), the pre-processor tries to create the composite grammar by combining the host grammar with the grammars of all specified EDSLs. Before that, the grammars are checked for possible conflicts. Currently, we build the union of all productions defined for the host language and all EDSLs. For example, the combined grammar for parsing TinySQL embedded in Groovy contains all productions defined in Listing 2 and Listing 9. As long as two EDSLs do not define two productions that have exactly the same terminals and non-terminals the pre-processor can distinguish them and will call the correct method of the homogeneous embedded DSL.

Parsing into forest of ASTs. When parsing (Step 4) the source code, we create a forest of all possible ASTs of the source code using the combined grammar. In the forest, every AST node is associated with the list of characters from the input stream and the corresponding production that has recognized the characters. This forest contains only valid ASTs, but due to ambiguities—in particular in the presence of island grammars—there are several possible ASTs under which the DSL file can be recognized by the parser.

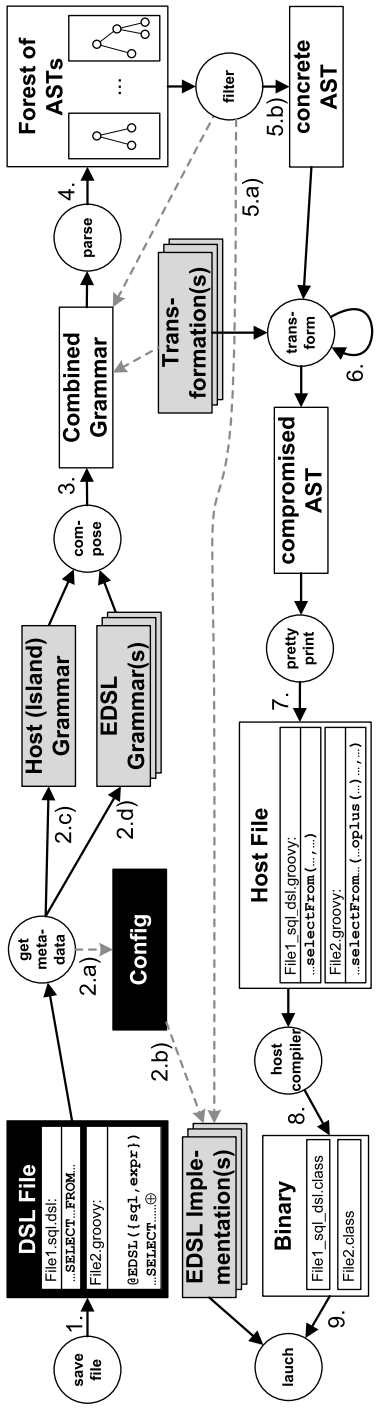

Fig. 3. Overview of the compilation process.

Selecting one AST in concrete syntax. We use *disambiguation filters* [17] to select the AST with the well-defined execution semantics. Therefore, we filter the forest of ASTs (Step 5) and select only one from it (Step 5 b). To find the right AST, we take both the syntactic and the semantic information into account that is defined in the EDSL interface.

The filtering always prefers AST nodes associated with detailed productions over AST nodes with water productions. Additionally, we perform semantic checks to eliminate invalid ASTs from the forest. For every subexpression passed to another expression, the filter determines the corresponding production and method in the EDSL interface via introspection (Step 5 a). This allows the filter to take into account the return type of the method that defines the semantic type of the computation returned by this subexpression.

Transforming the AST to host syntax. In the transformation step (Step 6), the pre-processor uses all transformations in the order they have been selected by the end user. The transformation rewrites every AST expression in concrete syntax into its compromised syntax. After that the EDSL program is comparable to a program that would have been written using host syntax.

Pretty printing, host compilation, and launching. After the AST has been transformed to the host language syntax (Step 7), the AST is pretty printed into a file that only contains valid host language expressions. The pre-processor uses the unchanged host compiler to compile this file to binary form (Step 8) and launches the binary (Step 9).

5.4. Enabling a new host language

To support a new host language, a language developer has to specify the parts of the host language's syntax that are relevant when embedding DSLs. This set contains the most important productions that are needed to syntactically recognize boundaries of basic program structures in the host language. This typically includes: its syntax layout (white spaces and delimiters), identifiers, type definitions, brackets, variable declarations, method calls, object instantiations, block structures, and so forth. These productions are required to support embedding DSL expressions into programs written in the host language.

To ease the definition of production rules, we provide special non-terminals. As discussed previously, we support the categories: `␣`, `?␣` and `!␣`. Additionally, *Id* is for identifiers that match the regular expression `[A-Za-z0-9_+]`. For the specification of partial grammar definitions, developers can use the water rule `$water$` that matches arbitrary characters but only when no other production matches.

For defining BNF-like custom productions, we provide an API. Language developers use this API to add new production rules. For example, in case of Groovy, the developer only uses the API to define the productions listed in Listing 9. As outlined previously, only the most important productions for recognizing the distinctive parts of Groovy's syntax need to be formalized. The grammar uses the *water rule* (cf. Section 2) in line 11 to avoid having to define the complete grammar of Groovy. Using the water rule allows statements of the host language and of the EDSL that have not been formalized to be parsed.

```
1  S_Groovy → Program
2  Program → Statements
3  Statements → Statement ?␣ Statements | Statement
4  Name → Id Id ?␣ "." ?␣ Name
5  Type → rType | pType
6  pType → Name | MethodCall | ConstructorCall
7  Arguments → Type | Arguments ?␣ "," ?␣ Type
8  Application → "(" ?␣ ")" | "(" ?␣ Arguments ?␣ ")"
9  MethodCall → Name ?␣ Application
10 ConstructorCall → "new" Name ?␣ Application
11 Statement → { ?␣ Statements ?␣ Statements } | $water$
```

Listing 9. Island grammar for Groovy.

Compared to approaches with a complete syntax definition, the integration of the host language provides fewer guarantees, because many expressions will be parsed using the water rule. However, the language developer can always specify more productions if desired. Conversely, the partial grammar requires far fewer productions to be formalized and is sufficiently precise for most practicable purposes. When developing a new embedded DSL in a host language, the developer imports the partial grammar for the language.

The pre-processing step performed by our approach requires that the developer provides three transformations which operate on the EDSL's AST: The first transformation rewrites every AST node with concrete syntax to an equivalent AST node in compromised syntax. The second transformation adds bootstrap logic into the AST that will evaluate the DSL statements. The third transformation pretty prints the AST into host language code.

Basically, all transformations are Java classes that are *visitors* [25] that implement the interface `ASTTransformation`. Each transformation can define a set of *requirements* on the AST under transformation. Before a transformation is applied, all its requirements are checked for by reading the meta-data that is attached to the AST. A transformation can then add, change, and remove AST nodes and the meta-data attached to these nodes. The transformation stores its *assurances* by adding meta-data to the AST. Requirements and assurances allow the plug-in to validate the chain of transformations.

6. Implementation

We have implemented the proposed approach as an Eclipse plug-in called TigersEye. Our pre-processor is an incremental project builder that can be integrated with every project's build process. In the following, we give a brief overview of TigersEye.

6.1. Implementing the pre-processor as a project builder

The developed pre-processor is host language independent. As a proof of concept, TigersEye supports two host languages: Groovy and Java. The only requirements of TigersEye are that the host language compiles to Java Bytecode and that it is possible to extract meta-data from an EDSL implementation, e.g., provided by means of annotations. These requirements are met by most modern languages, such as C#, Scala, Smalltalk.

Using the current version of TigersEye, language developers can develop embedded DSLs (EDSLs) using Groovy or Java. Using *Java reflection*, we introspect EDSLs to extract the EDSL's meta-data, such as information about the production rules and the transformations. The extracted partial grammar is made accessible as a set of first-class objects. When a program uses several EDSLs, the pre-processor first composes the EDSLs' grammars and meta-data.

If multiple EDSLs are used, TigersEye always creates the union of grammars of the composed languages. To resolve syntactic conflicts we have implemented disambiguation filters [17]. Our evaluation has shown that this support is sufficient to resolve syntactic conflicts.

Our parser implementation uses the Earley algorithm [16], which can handle all types of CFGs. It directly uses the grammar of the EDSL and host language as supplied by the language developer. An important advantage is that the generic parser makes the grammar available as a first-class object during the transformation process. This enables the pre-processor to extend and compose grammars of the host language and of the embedded DSLs.

Our implementation of the Earley algorithm adds support for island grammars [10] as outlined next. The standard Earley algorithm has three steps that are continuously repeated for every character, namely (1) scan, (2) predict, and (3) complete. To support island grammars, we extended the scan step. The new scan step first tries to accept the next character using the explicitly defined productions. Only if no production accepts the next character, we use the water rule to recognize it as *water*. Each parse step that is stored as a so-called Earley item in the parser chart is associated with the character recognized in the step. After parsing, the chart provides all information for the full *forest of ASTs*.

To represent ASTs and to implement term rewriting, we use *ATerms* [26]. ATerms unify several term formats, optimize storage through *maximal subterm sharing*, and allow language-neutral exchange. To implement custom transformations—as extensions of our pre-processor—the language developer also has to use ATerms. The transformation steps are: (1) the pre-processor translates the Earley chart to ATerms, (2) then the pre-processor uses the ATerms' pattern matching facilities to perform transformations on the ASTs, after that (3) the pre-processor uses the ATerm's ASTs to generate the host language code.

Note that the project builder outputs generated host language code into a special folder, e.g. by default "`./tigerseye-bin`". From this folder, the Groovy or Java compiler will further process the generated code in order to compile it to Java bytecode into the standard "`./bin`" folder.

6.2. Implementing convenience IDE services

For convenient usage of TigersEye, we have implemented a set of convenience IDE services which are discussed in the following. In general, an end user first has to download and install an EDSL implementation. EDSL implementations are generally deployed as Eclipse plug-ins that extend a well-defined extension point. After installation, the language implementation is registered with TigersEye's EDSL registry and can be configured using a generic (language independent) Eclipse preferences pane. It is, e.g., possible to (de)activate an EDSL.

6.2.1. Syntax highlighting

TigersEye provides an editor that supports syntax highlighting for source files that use one or more EDSLs. Before opening a file, the editor first extracts the set of used languages. The editor automatically detects the used languages if the source file name's extension, which is encoded as "`<filename>.<edsl-key>.dsl`", contains the unique key of the corresponding EDSL. Alternatively, if the EDSL program uses the special `@EDSL(<list of EDSLs>)` annotation, the editor extracts the set of EDSLs' keys from this annotation. Next, it loads all selected EDSLs and extracts all reserved keywords of these languages. Finally, when opening the file, the editor highlights all corresponding keywords.

Using the registry, the end user can further configure the syntax highlighting.

Table 1
Number of grammar rules that need to be specified to provide the concrete syntax.

EDSL	Complete grammar	Island grammar	Savings (%)
TinySQL	6	1	83
BNF	9	9	0
State machines	10	2	80
Simple logo	11	1	91
JSON	27	8	70

6.2.2. Launching

For launching an EDSL program, an end user can invoke "Runs as…" from Eclipse. The launcher ensures that the builder has pre-processed all EDSL files and will run the pre-processed and compiled EDSL program.

6.2.3. Debugging

Currently, TigersEye only provides basic support for debugging EDSLs with compromised syntax. It is only possible to consider "library" calls as statements in the language and to step over them during debugging.

In the implementation of the current Eclipse prototype, debugging is only supported for EDSLs that do not use concrete syntax. The pre-processing step necessary in case of EDSLs with concrete syntax is not compatible with the current debugger implementation. The necessary information that relates some piece of EDSL code with the code created by its transformation to host syntax is not yet available to the debugger. We are, however, confident that in the future we can extend the debugger to extract the necessary information from the compiled files and the EDSL's meta-data to fully support debugging of EDSLs code.

7. Evaluation

We have evaluated our approach w.r.t. four aspects. Section 7.1 measures the costs for EDSL grammar specifications. Section 7.2 measures the costs for adding support for new host languages. Section 7.3 validates the approach by means of case study. Finally, Section 7.4 discusses the quality of the tooling.

7.1. Costs of EDSL grammar specification

To evaluate our approach, we have implemented several existing and experimental DSLs. We always started by implementing a basic version of the DSLs without support for concrete syntax. After that, we incrementally added annotations to their implementations to support the domain's syntax.

We have implemented the following four small DSLs: (1) a BNF dialect to specify CFGs, (2) TinySQL, (3) a DSL for defining state machines, (4) some dialects of the Logo language for teaching programming. As a real-world example, we have also implemented support for JSON (Java Script Object Notation).[7] The latter DSL currently supports values with the following types: string, array or object.[8] In all cases, we were able to provide the concrete syntax of the particular domain. Furthermore, the effort necessary to provide the concrete syntax for an EDSL is practically eliminated as shown in Table 1. For example, to provide the concrete syntax for TinySQL, it was sufficient to specify a single grammar rule (production) while a complete specification would require the specification of six productions. We expect that for other—in particular larger—languages even more savings can be expected if it is possible to piggyback on the concrete syntax definitions of the host language.

7.2. Costs for new host languages

The integration of the EDSLs with the host languages also requires only negligible effort. For Java and Groovy, it was sufficient to specify only 19 productions each. As elaborated in Section 5.4, to parse the host languages with island grammars, we had to define only the most important productions that recognize the host languages' basic syntactic structure (see Table 2).

7.3. Case study — Embedding JSON into Groovy

To validate our approach, we have embedded most of JSON (cf. Section 7.1) into Groovy. JSON is a human-readable data exchange format that is a strict subset of the JavaScript programming language. For example, Listing 10 shows the definition of an anonymous object with four properties (key/value pairs). We have chosen JSON, because it is a non-trivial real world language that is widely used.

[7] http://www.JSON.org/.

[8] Our current implementation could be extended to support more types.

Table 2
Number of grammar rules that need to be specified to enable host language integration.

Host language	Complete grammar	Island grammar	Savings (%)
Java	1700	19	98.9
Groovy	950	19	98.0

```
1  {
2   "Title" : "Groovy in Action" ,
3   "Authors" : [ "Dierk Konig" , "Andrew Glover" , "Paul King" ,
4                 "Guillaume Laforge" , "Jon Skeet" ] ,
5   "Publisher" : "Manning" ,
6   "ISBN" : "9781935182443"
7  }
```

Listing 10. A JSON object.

To get a first idea of the effort necessary to embed JSON using our approach, we compare the effort required to specify the grammar using island grammars with the effort using a standard formalism such as BNF. As shown in Listing 12, using our island-grammars-based approach requires the specification of only six `@DSLMethod` annotations when compared to the 27 productions that are necessary in case of the standard JSON grammar (cf. Listing 11).

```
1  S_JSON → object
2  object → "{}" | "{" members "}" //2 productions
3  members → pair | pair "," members //2 productions
4  pair → string ":" value
5  array → "[]" | "[" elements "]" //2 productions
6  elements → value | value "," elements //2 productions
7  value → string | object | array //3 productions
8  string → """" | """ chars """
9  chars → char | charchars //2 productions
10 char → ... //assuming 1 production that recognizes 1.1 Mio Unicode v6.1 chars
11 char → ... //9 productions for control chars
```

Listing 11. The JSON grammar [27].

Furthermore, embedding JSON into a host language is not without challenges and these challenges can be regarded as representative for those challenges that arise when reasonably large languages are embedded into some host language. As shown in Listing 11, JSON's grammar is small but makes heavy use of characters—in particular curly braces and square brackets—that are also used by many potential host languages. Such situations can lead to complex syntactic ambiguities. For example, in Line 4 of Listing 13 a variable (book) is defined which is assigned a JSON object. However, while parsing the program this cannot immediately be decided. The opening curly bracket in line 4 could be the beginning of a definition of a JSON object or a Groovy closure. To correctly disambiguate such situations, the syntactic categories of the surrounding expressions need to be taken into account. In this case the enclosed sub-expressions are all JSON expressions and therefore the enclosing curly brackets are considered as belonging to the JSON DSL. After disambiguation, JSON expressions are rewritten to method calls that construct a JSON object in compromised syntax.

7.4. Quality of the tooling

Furthermore, the effort that is necessary to support concrete syntax for a homogeneous embedded EDSL that can be used across different host languages is reduced. In case of the four discussed DSLs, no effort was necessary to directly support concrete syntax for the discussed DSLs when we embedded them in Java (they were first embedded into Groovy.) This, however, requires that we can reuse the same DSL interpreter for the new target host languages and that support for the target languages is readily available in our pre-processor. If this is the case, it is in general only necessary to adapt the EDSL's grammar to the new host language grammar. Due to our use of island grammars and the syntactical similarities between Groovy and Java, no adaptation was necessary.

Though our current approach already provides good support for incrementally developing embedded DSLs with concrete syntax, the performance is not yet optimized. The runtime requirements of the used Earley algorithm are already high and the added support for island grammars increases them further. Currently, parsing and transforming DSL programs of 500 effective lines of code takes approx. 10 s.

```
@DSLClass(whitespaceEscape = " ") // does not count as a production
public class JsonDSL extends Interpreter implements IJsonDSL {
    ...
 @DSLMethod(production = "{ }")
 public Map object() { return new HashMap(); }

 @DSLMethod(production = "{ p0 }")
 public Map object(JsonMap members) { return members.toMap(); }

 @DSLMethod(production = "p0 : p1", topLevel=false)
 public JsonMap pair(String string, Object value) {
   return new JsonMap(string, value);
 }

 @DSLMethod(production = "p0 : p1 ,  p2",  topLevel=false)
 public JsonMap pairs(String string, Object value, JsonMap pairs) {
   return new JsonMap(string, value, pairs);
 }

 @DSLMethod(production = "[ ]")
 public List  array(JsonArray elements) {
   return new LinkedList();
 }

 @DSLMethod(production = "[ p0 ]")
 public List  array(JsonArray elements) {
   return Arrays.asList(elements.toArray());
 }

 @DSLMethod(production = "p0", topLevel=false)
 public JsonArray array(Object value) {
   return new JsonArray(value, null);
 }

 @DSLMethod(production = "p0 , p1", topLevel=false)
 public JsonArray elements(Object value, JsonArray elements) {
   return new JsonArray(value, elements);
 }
}
```

Listing 12. Excerpt of the JSON implementation.

```
def closure = { x -> x * x }
println "Result of execution closure: " + closure.call(5);  // prints 25

def book = {
  "Title" :  "Groovy in Action" ,
  "Authors" : [  "Dierk Konig" ,  "Andrew Glover" , "Paul King" ,
                 "Guillaume Laforge" , "Jon Skeet"  ]  ,
  "Publisher" :  "Manning" ,
  "ISBN" :  "978-1932394849"
}
println book; //prints book as string
```

Listing 13. A JSON object embedded into Groovy code.

8. Comparison with related work

Table 3 shows a comparison of our approach with related work on EDSLs, which will be further elaborated in the next section. We compare the embedding approaches w.r.t. the host languages that are supported out of the box, if the approach is host language independent, and, if so, if the grammar of the complete host language has to be available. Furthermore, we compare the "embedding style" and if the approach supports "concrete syntax", whether a complete specification of the DSL's grammar is required. Finally, we compare the class of languages supported by the approach. We classify our approach as a *hybrid* embedding approach. Since TigersEye is a pre-processor that uses homogeneous embedded DSLs.

Table 3
Comparison language embeddings' support for concrete syntax.

	Host language			Embedding					
Approaches…	Supported Host Languages	Host Language Independent	Grammar Size for Host Language	Embedding Style	Concrete Syntax	Grammar Size for Embedding	Domain-Specific Tool Support	Separate Compilation	Supported Languages
…with compromised syntax									
Pure embedding [4]	Haskell			homo.	no			✓	
Embedded compiler [28]	ML			hetero.	no			✓	
Jargons [29]	Scheme			homo.	no				
DSL2JDT [30]	Java			homo.	no			✓	
Polymorphic embedding [31]	Scala			homo.	no			✓	
Reflective embedding [22]	Groovy			homo.	no			✓	
…with concrete syntax									
MetaBorg [6]	Java, …	✓	cmpl.	hetero.	**yes**	cmpl.	✓		**CFG**
TXL [7]	Java, …	✓	cmpl.	hetero.	**yes**	cmpl.			LL(*)
Converge [2]	Converge			homo.	**yes**	cmpl.			**CFG**
π [8]	π			homo.	**yes**	cmpl.			**CFG**
Helvetia [9]	Smalltalk			homo.	**yes**	cmpl.	✓		PEG
SugarJ [32]	Java		cmpl.	homo.	**yes**	cmpl.	✓		**CFG**
TigersEye	Java, Groovy, …	✓	**prt.**	**hybrid**	**yes**	**prt.**	✓	✓	**CFG**

8.1. EDSL approaches with compromised syntax

The homogeneous EDSL approaches *pure embedding* [4], *Jargons* [29], *DSL2JDT* [30], *polymorphic embedding* [31], and *reflective embedding* [22] do not use meta-programming and do not support concrete syntax.

The heterogeneous EDSL approach of *embedded compilers* [28] uses meta-programming in the ML language. However, for end user programs, there is no support for concrete DSL syntax, because programs must be encoded in ML.

A common feature of all above approaches except Jargons is that they support separate compilation. Hence, it is possible to independently develop the EDSLs and to distribute them in binary form as in case of TigersEye.

8.2. Heterogeneous EDSL approaches with support for concrete syntax

Heterogeneous approaches, e.g., MetaBorg [6] and *TXL* [7], first supported the embedding of DSLs with concrete syntax. Compared to TigersEye both, however, require that developers provide the language implementations as source code and therefore do not support separate compilation.

MetaBorg [6] is the most mature approach with respect to embedding DSLs with concrete syntax [2]. MetaBorg uses Stratego/XT to specify syntax definitions and AST transformations. Given the complete syntax definition of the EDSL and the host language, MetaBorg generates a pre-processor. This pre-processor compiles programs in concrete syntax to host language programs that use a library that implements the DSL's functionality. MetaBorg is *heterogeneous*. Stratego/XT supports island grammars [23], but MetaBorg does not use them for defining grammars of EDSLs. Like our approach, MetaBorg is host-language independent, but currently only supports Java. For example, to support Groovy the complete Groovy grammar has to be (re-)written in SDF which corresponds to approximately 950 of our BNF-like productions (this estimation is based on a detailed analysis of the Groovy ANTLR grammar). To enable parsing Java and Groovy files with embedded DSL syntax we only needed to specify a few productions for Java resp. Groovy. In contrast to TigersEye, the definition of the DSL's concrete syntax and semantics uses different languages – the Stratego language and the host language. This heterogeneity makes MetaBorg less uniform than TigersEye.

TXL [7] is an heterogeneous embedding approach that can rewrite embedded programs into any target language similar to MetaBorg. However, TXL only supports LL(*) grammars and there is no special tool support.

8.3. Homogeneous EDSL approaches with support for concrete syntax

Next, we discuss homogeneous embedding approaches that use special host languages to support concrete syntax: *Converge* [2], π [8], *Helvetia* [9], and *SugarJ* [32]. A common difference of all three approaches compared to TigersEye is that these approaches always require language developers to define the complete syntax of each EDSL. However, a commonality is that all approaches support multiple embeddings.

Converge uses compile-time meta-programming to enable access to AST nodes in programs. Programs are allowed to rewrite themselves during compilation. The language developer specifies the complete grammar in a BNF-like DSL – from which an Earley parser is generated. The developer uses Converge to implement a transformer that rewrites AST nodes with concrete syntax to plain Converge expressions. Converge does not provide specialize tool support for DSL syntax and it does not support separate compilation.

The π language uses runtime meta-programming that enables programs to access their AST at runtime. The developers define all DSL expression types as so-called *patterns*. Each pattern recognizes a piece of the concrete syntax and gives it a *meaning* – an interpretation in the π language. In π, there is neither tool support nor separate compilation.

In Helvetia [9], Smalltalk is used to create homogeneous embedded DSLs. The language developer uses Smalltalk to implement a complete parser using a parser combinator library and to specify rewrite patterns on AST nodes. Helvetia parses DSL code and rewrites it to Smalltalk code.

SugarJ [32] extends the Java language with a special *import statement* to import the concrete syntax of an embedded DSL. In SugarJ, developers define language extensions for Java in the Stratego language, and there is a special Eclipse plug-in that rewrites EDSL programs with a concrete syntax to Java plain code. SugarJ has the same restrictions as MetaBorg, and therefore does not support separate compilation.

Converge, π, Helvetia, and SugarJ are not host language independent, thus they cannot be used to integrate EDSLs into other host languages than their own host language. By relying on special features of their host language, term rewriting and DSL execution is homogeneously integrated. In contrast, TigersEye does not require special host language features. We use a pre-processor for transformation and for bootstrapping the EDSL syntax and semantics into the host program.

8.4. Language workbenches

There are various language workbenches that target expert developers for implementing languages with tool support in a special environment. For Java, there are *JetBrains Meta-Programming System (MPS)* [33], *openArchitectureWare* [34], *IDE Metatooling Platform (IMP)* [35], and *Spoofax* [36]. For C#, there is *Intentional Software* [37]. With respect to providing tools for languages, workbenches have a similar motivation like EDSLs. However, in contrast to EDSLs, workbenches require the language developer to follow an approach-specific process to define syntax and semantics. TigersEye has a different focus since it targets programmers who do not have to be experts in a certain syntax formalism. Developers do not need to learn a new meta-language, they only need to know the object-oriented programming language and our conventions (i.e. annotations), which only define new rules inside the OO formalism and in one uniform model. Unlike many workbenches that are heterogeneous and non-uniform, TigersEye supports homogeneous EDSLs that are uniform. Because the host compiler is no external tool, there is no conceptual gap between EDSLs and host.

Among these approaches, Spoofax is most relevant because it provides explicit meta-languages for combining multiple languages in a way that resembles composing EDSLs in TigersEye. In Spoofax [36], developers implement languages using the Stratego language. Given a language's implementation Spoofax then generates Eclipse plug-ins to provide specialized IDE support. However, for every new combination of languages, a language developer must generate a new plug-in. A limitation of the missing support for separate compilation is that, when there are multiple language combinations that use the same constituent languages, the plug-in generator repetitively generates and compiles the same code for the constituent languages. Although generated code of the constituent languages could be re-used, Spoofax currently does not support this. Generating language-specific code in a way such that it can be re-used in different language combinations would require fundamentally changing the implementation of the way Spoofax generates code [38]. More severely, when the number of used languages grow, the size of the generated code grows exponentially. This can lead to plug-in binaries with the size of a Gigabyte and more [39]. In TigersEye, the size of the binary code grows liberally with the number of used languages.

Jetbrains MPS also supports combining multiple languages, but is otherwise quite different from our approach. Specifically, developers do not design text-based languages. Instead, developers have to use a visual cell-based editor which maps programs to executable code, e.g. to Java. There is little information available about MPS, which makes it hard to compare.

8.5. Extensible compilers

Extensible compilers, such as *Polyglot* [40] and *JastAdd* [41] could also be used to integrate a DSL in a host language. However, in contrast to TigersEye they have a different focus. First, they focus on language extensions for a particular base language. TigersEye on the other hand focuses on embedded DSLs that will be integrated with a host language as an extension and where the host language may not even be designed or implemented with extensibility in mind. Second, instead of application developers, they target compiler experts. In general, extensible compilers have a lack of tool support, thus they do not support syntax highlighting or other IDE services.

8.6. Grammar inference

Grammar inference algorithms generally build grammars by either analyzing a set of existing programs or by repetitively asking an oracle. Hrncic et al. [42], for example, use a grammar inference algorithm that uses evolutionary learning combined

with local search to derive EDSL grammars from positive and negative examples. As in our case, the derived grammar uses island grammars as the underlying technique and the overall goal is also to minimize the costs associated with creating a grammar specification. However, when compared to their proposal, we address the expression of the EDSL's syntax and also the implementation of the EDSL's semantics. The latter in particular facilitates resolving syntactic ambiguities using type information.

9. Conclusions

In this paper, we have presented a new approach that significantly reduces the effort necessary to enable concrete syntax for embedded DSLs. The basic idea is to enable language developers to incrementally specify a DSL's syntax using island grammars. When compared to related work, the main advantage of our approach is that only parts of a DSL's concrete syntax need to be specified. This has four main implications: First, it is only necessary to specify a DSL's concrete syntax for those parts where the host language's concrete syntax conflicts with the domain syntax. This reduces the effort necessary for specifying the DSL's grammar. Second, our approach supports the incremental definition of a DSL's concrete syntax, which helps to reduce the initial effort necessary when developing an EDSL. Third, the implementation of an embedded DSL can be—to a very large degree—host-language independent. This facilitates reuse of EDSL implementations if the host language compiles to a bytecode representation such as Java or CIL bytecode. Fourth, developers can separately compile EDSL implementations and distribute them as bytecode. Later on, end users can download the binary versions of EDSLs and compose them.

To evaluate the approach, we implemented an Eclipse project builder to embed several DSLs into Java and Groovy and compared the costs with related work.

In future work we will try to address the performance issues related to the use of an Earley parser with support for island grammars. Further, we plan to make the annotations more practicable, e.g. by using *type annotations* (JSR 308) and by learning lessons from other non-embedded approaches that use annotations [43]. Currently, we are also working on syntactic and semantic analyses of DSLs to provide special guarantees. Better debugging of EDSLs with concrete syntax is also a possible direction for future work.

Acknowledgments

This work was partly supported by the EMERGENT project (01IC10S01N), Federal Ministry of Education and Research (BMBF), Germany. We would like to thank Kamil Erhard, Leo Roos, Pablo Hoch, and Yahya Benkaouz for their valuable contribution to the implementation of TigersEye. Furthermore, we also like to thank Martin Monperrus and Andreas Sewe for valuable comments.

References

[1] M. Mernik, J. Heering, A.M. Sloane, When and how to develop domain-specific languages, ACM Computing Surveys (CSUR) 37 (4) (2005) 316–344.
[2] L. Tratt, Domain specific language implementation via compile-time meta-programming, ACM Transactions on Programming Languages and Systems 30 (6) (2008) 1–40.
[3] J. Melton, A. Eisenberg, Understanding SQL and Java Together: A Guide to SQLJ, JDBC, and Related Technologies, Morgan Kaufmann, 2000.
[4] P. Hudak, Building domain-specific embedded languages, ACM Computing Surveys 28 (4) (1996) 1–6.
[5] T. Kosar, M. López, E. Pablo, P. Barrientos, M. Mernik, A preliminary study on various implementation approaches of domain-specific language, Information and Software Technology 50 (5) (2008) 390–405.
[6] M. Bravenboer, E. Visser, Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions, in: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA'04, ACM, New York, NY, USA, 2004, pp. 365–383.
[7] J.R. Cordy, The TXL source transformation language, Science of Computer Programming 61 (3) (2006) 190–210 (special issue on The Fourth Workshop on Language Descriptions, Tools, and Applications (LDTA'04)).
[8] R. Knöll, M. Mezini, π: a pattern language, ACM SIGPLAN Notices 44 (10) (2009) 503–522.
[9] L. Renggli, T. Gîrba, O. Nierstrasz, Embedding languages without breaking tools, in: T. D'Hondt (Ed.), European Conference on Object-Oriented Programming, in: Lecture Notes in Computer Science, vol. 6183, Springer, Berlin Heidelberg, 2010, pp. 380–404.
[10] L. Moonen, Generating robust parsers using island grammars, in: Proceedings of the 8th Working Conference on Reverse Engineering, IEEE Computer Society, 2001, pp. 13–22.
[11] E. Visser, Syntax definition for language prototyping, Ph.D. Thesis, Universiteit van Amsterdam, The Netherlands, 1997.
[12] D. Knuth, Semantics of context-free languages, Theory of Computing Systems 2 (2) (1968) 127–145.
[13] B. Ford, Parsing expression grammars: a recognition-based syntactic foundation, ACM SIGPLAN Notices 39 (1) (2004) 111–122.
[14] E. Visser, Scannerless generalized-LR parsing, Tech. Rep. P9707, Programming Research Group, University of Amsterdam, July 1997.
[15] A. Johnstone, P. Mosses, E. Scott, An agile approach to language modelling and development, Innovations in Systems and Software Engineering 6 (2010) 145–153.
[16] J. Earley, An efficient context-free parsing algorithm, Communications of the ACM 13 (2) (1970) 94–102.
[17] M. van den Brand, J. Scheerder, J. Vinju, E. Visser, Disambiguation filters for scannerless generalized LR parsers, in: R. Horspool (Ed.), Compiler Construction, in: Lecture Notes in Computer Science, vol. 2304, Springer, Berlin Heidelberg, 2002, pp. 143–158.
[18] N. Synytskyy, J.R. Cordy, T.R. Dean, Robust multilingual parsing using island grammars, in: Proc. of Conference of the Centre for Advanced Studies on Collaborative Research, CASCON'03, IBM Press, 2003, pp. 266–278.
[19] M. Mernik, M. Lenic, E. Avdicausevic, V. Zumer, Multiple attribute grammar inheritance, Informatica 24 (2) (2000) 319–328.
[20] M. Mernik, V. Zumer, Incremental programming language development, Computer Languages, Systems & Structures 31 (1) (2005) 1–16.
[21] P. Hudak, Modular domain specific languages and tools, in: P. Devanbu, J. Poulin (Eds.), International Conference on Software Reuse, 1998, pp. 134–142.
[22] T. Dinkelaker, M. Eichberg, M. Mezini, An architecture for composing embedded domain-specific languages, in: Aspect-Oriented Software Development, ACM, 2010, pp. 49–60.

[23] L.C. Kats, M. de Jonge, E. Nilsson-Nyman, E. Visser, Providing rapid feedback in generated modular language environments: adding error recovery to scannerless generalized-LR parsing, Object-Oriented Programming, Systems, Languages, and Applications (2009) 445–464.
[24] P. Maes, Computational reflection, Ph.D. Thesis, Vrije Universiteit, Brussel, 1987.
[25] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional, 1995.
[26] M. van den Brand, P. Klint, ATerms for manipulation and exchange of structured data: it's all about sharing, Information and Software Technology 49 (1) (2007) 55–64.
[27] D. Crockford, Json.org homepage, http://www.json.org/ (2006).
[28] S.N. Kamin, Research on domain-specific embedded languages and program generators, Electronic Notes in Theoretical Computer Science 14 (0) (1998) 149–168.
[29] F. Peschanski, Jargons: experimenting composable domain-specific languages, in: Workshop on Scheme and Functional Programming, Firenze, Italy, 2001, pp. 42–46.
[30] M. Garcia, Automating the embedding of domain specific languages in eclipse JDT, July 2008. http://www.eclipse.org/articles/Article-AutomatingDSLEmbeddings/.
[31] C. Hofer, K. Ostermann, T. Rendel, A. Moors, Polymorphic embedding of DSLs, in: Generative Programming and Component Engineering, 2008, pp. 137–148.
[32] S. Erdweg, T. Rendel, C. Kästner, K. Ostermann, SugarJ: library-based syntactic language extensibility, in: Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), OOPSLA'11, ACM, New York, NY, USA, 2011, pp. 391–406.
[33] S. Dmitriev, Language oriented programming: the next programming paradigm, JetBrains onBoard 1 (2) (2004).
[34] T. Stahl, M. Völter, C. Czarnecki, Model-Driven Software Development, John Wiley & Sons, England, 2006.
[35] P. Charles, R.M. Fuhrer, S.M. Sutton Jr., IMP: a meta-tooling platform for creating language-specific IDEs in eclipse, in: Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering, ASE'07, ACM, New York, NY, USA, 2007, pp. 485–488.
[36] L.C.L. Kats, E. Visser, The Spoofax Language Workbench. Rules for Declarative Specification of Languages and IDEs, in: M. Rinard (Ed.), Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA'10, 2010, pp. 444–463.
[37] C. Simonyi, M. Christerson, S. Clifford, Intentional software, in: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, Object-Oriented Programming, Systems, Languages, and Applications'06, ACM, New York, NY, USA, 2006, pp. 451–464.
[38] Visser, Personal communication (June 2011).
[39] S. Erdweg, L.C.L. Kats, T. Rendel, C. Käastner, K. Ostermann, E. Visser, Growing a language environment with editor libraries, in: Proceedings of Conference on Generative Programming and Component Engineering, GPCE, ACM, 2011, pp. 167–176.
[40] N. Nystrom, M. Clarkson, A. Myers, Polyglot: an extensible compiler framework for Java, in: G. Hedin (Ed.), Compiler Construction, in: Lecture Notes in Computer Science, vol. 2622, Springer Berlin Heidelberg, 2003, pp. 138–152.
[41] T. Ekman, G. Hedin, The JastAdd extensible Java compiler, in: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications, OOPSLA'07, ACM, New York, NY, USA, 2007, pp. 1–18.
[42] D. Hrncic, M. Mernik, B.R. Bryant, Embedding DSLs into GPLs: a grammatical inference approach, Journal of Information Technology and Control 40 (4) (2011) 307–315.
[43] J. Poruban, M. Forgac, M. Sabo, M. Behalek, Annotation based parser generator, Computer Science and Information Systems 7 (2010) 291–307.