On Embedding Domain-specific Languages with User-friendly Syntax

Gilles Dubochet

École polytechnique fédérale de Lausanne, Switzerland

Abstract

I present ZyTyG, a strategy to define domain-specific languages with a user-friendly syntax. This strategy does not requires any pre-processor or meta-programming facility and only relies on features provided by the object-oriented host language. I describe its application to ScalaDBC, a database library with an embedded DSL to encode queries in SQL-like syntax and that uses this strategy.

1 Introduction

Domain-specific languages (DSL) [10, 6] can be implemented as stand-alone languages with a native interpreter or compiler. Since this is time-consuming, DSL designers often prefer extending an existing general-purpose language with the domain-specific additions — the DSL is "embedded" in the host language and called a "DSEL" [5]. Embedding a DSL is not trivial. Most languages support easy extension by importing a library of pre-written code. Changing the semantics or syntax, on the other hand, is difficult: DSEL designers often use generative programming such as meta-programming [9, 4], C++ template macros [1, 4] or custom pre-processors [3, 2] to rewrite the domain-specific syntax.

This paper describes "zygotic type-directed growth" (ZyTyG), a new strategy to embed DSLs with a custom syntax and without using generative programming. The syntax allowed for the DSEL is limited, but is well suited for user-friendly DSLs that use many "decorative" tokens to improve legibility. SQL is a typical example of such a DSL. The ZyTyG host language must be object-oriented and support a small set of behaviours (in particular implicit coercions, also called views) found in some statically-typed languages.

This strategy is then used to implement ScalaDBC, a database library with a DSEL to encode queries in SQL-like syntax. ScalaDBC is implemented in the Scala programming language [8, 7], an object-oriented language suitable for applying ZyTyG.

The rest of the paper is organised as follows. Section 2 presents the key ideas of ZyTyG. Section 3 presents ScalaDBC, a real-life library based on ZyTyG. Section 4 describes some research directions that might lead to broaden ZyTyG's applicability.

2 ZyTyG

ZyTyG is the strategy I propose for designing DSELs in object-oriented languages. The three key ideas are outlined below.

Idea 1: strings of identifiers. This idea requires the host language to support method-as-infix-operator syntax. This allows writing expressions like "$\mathtt{e}_1\ \mathtt{op}_a\ \mathtt{e}_2\ \mathtt{op}_b\ \mathtt{e}_3$" where "`e`" are expressions of the host language, and "`op`" are operators. These are converted to chains of method calls like "$\mathtt{(e}_1\mathtt{.op}_a\mathtt{(e}_2\mathtt{)).op}_b\mathtt{(e}_3\mathtt{)}$". Operators, effectively methods, are any legal method identifier in the host. Expressions are any host construct, including name identifiers. This allows writing any string of identifiers, a first step towards embedding a DSL.

Idea 2: building domain-specific data. The first host expression in a string of identifiers serves as a "zygote" from where the rest of the expression will grow. This host expression is an instance of a zygote class whose supported methods have the names of all tokens allowed immediately to the right of it.

As an example, I will use the *SetDSL* language. *SetDSL* has expressions like "`set1 join set2`" or "`set1 union set2`" where "`join`" and "`union`" are keywords. "`set1`" is the zygote and is a member of the following class in the host language's perspective.

```
class SetZygote {
  SetZygote union (SetZygote right) {...}
  SetZygote join (SetZygote right) {...}
}
```

Since the two methods return a "`SetZygote`" also, one can chain further operators to write expressions like "`(set1 join set2) union set3`".

This provides for constant syntax, where "`union`" and "`join`" can be used, from a syntactic point of view, interchangeably as operators. Richer syntax requires new zygote classes for every point in the expression that accepts a different set of tokens immediately on its right. As for the first zygote, these classes will define for each right-hand token a method named like the token. By selecting the appropriate zygote class as the return type for those methods, the tokens allowed immediately right of the *next* zygote are chosen. In other words, the syntax of the DSEL (that is the order in which tokens can be used) is defined by a type-directed path from the leftmost zygote through zygote methods.

For example, *SetDSL* is extended to define orderings on set data like "`select randomised set1 join set2`" where the ordering always starts the expression. This can be encoded in the host by defining the following.

```
class SelectZygote {
  SetZygote ordered (SetZygote right) {...}
  SetZygote randomised (SetZygote right) {...}
}
SelectZygote select = new SelectZygote()
```

The methods' return value means that the rest of the expression will use the syntax defined in "`SetZygote`".

Idea 3: from general-purpose to domain-specific, and back again. This idea requires views to be available in the host language. A view is a function that converts values of some type to values of another. An expression "`x.f(y)`" where either

1. the type of "`y`" is not compatible with the expected parameter for "`f`" or
2. the type of "`x`" does not support method "`f`"

will have the compiler automatically apply a view respectively to "`y`" and "`x`" if it allows successful typing.

In the host language's perspective, a DSEL expression is an instance of its last zygote class. Executing it should return a value that can be used as a normal value in the host language. The reverting view converts zygotes that can end the DSEL expression to native host values, executing the DSEL expression. If the host expression refers to the result, it will be as a host-language value, triggering the application of the view and executing the DSEL expression.

For example, say that the *SetDSL* expressions now return the host language's sets that contain an "`iterator`" method. "`(select ordered set1).iterator`" is a valid expression that demonstrates the seamless transition from DSEL to host. The compiler will insert a call to the reverting view around the *SetDSL* expression just before "`iterator`" is called.

On the other hand, if an expression *in* the DSEL expression is a value of the host language, it is not a zygote required to build-up the expression. Views from relevant host language values to the corresponding zygote are defined to convert them from the host language universe to the DSEL's universe.

For example, consider that the sets in *SetDSL* really are database tables defined by their name (a string) like in "`select ordered "table1" join "table2"`". However, since "`join`" is not defined on strings, "`"table1"`" must not be a string. A view from string to "`SetZygote`" will automatically be applied to convert the string to a zygote.

The mechanism used in zygotes to build-up the data structure used by the reverting view is beyond the scope of this article. The exact mechanism depends on the complexity of the DSL being parsed. A simple language might build-up the value after each operator call. A more complex language might require building a token-stream first and then parsing it using standard compiler techniques when the reverting view is applied.

Of course, ZyTyG is too simple not to be limited. In particular the fact that every second identifier must be a static method name is a major limitation. If at a given expression point a list of expressions is expected, they must either be explicitly encoded as host language lists, or separated by operators such as "`and`". Another limitation is that identifiers must be legal method names in the host. In particular, they cannot be tokens of the host language. That means many of the "good" identifiers — `while`, `if`, `class`, `new`, etc. — are already taken.

3 The ScalaDBC library

ScalaDBC is a wrapper around the JDBC database library and part of the Scala standard library. Unlike JDBC, ScalaDBC has to keep queries as native data structures — as opposed to strings. SQL's syntactic elements are bound to classes whose members are the sub-elements, effectively building a syntax tree for queries. However, writing this directly is verbose and unintuitive. For example, the query "`SELECT * FROM persons`" looks as follows:

```
statement.Select {
  val setQuantifier = None
  val selectList = Nil
  val fromClause = List(statement.Relation {
    val tableName = "persons"
    val tableRename = None
  })
  val whereClause = None
  val groupByClause = None
  val havingClause = None
}
```

A small library — effectively a ZyTyG DSEL — allows writing queries with SQL syntax. The previous query is then written as "`select () from "persons"`".

ScalaDBC support a considerable subset of SQL (1999 ISO), most of it with the DSEL syntax. This allows writing queries such as:

```
select (
  "age" is smallint,
  "name" is clob,
  "salary" is int )
from ("persons" naturalJoin "employees")
where (
  "gender" == "M" and
  ("city" == "lausanne" or "city" == "geneva"))
orderBy "age"
```

4 Open Issues

Exactly what syntactic tricks a language should support for hosting DSELs is an open issue. I have shown that method-as-infix-operator is limited, but sufficient for many realistic situations. Improving the set of syntactic tricks might make the ZyTyG strategy practical for general DSEL development.

Using views to improve DSELs is an exciting idea. Unfortunately, views make debugging more difficult. Since views are applied transparently by the compiler, the semantics of the executed expression differ from that of the original expression. It becomes worse with malformed DSEL expressions: zygotes, which are certainly not designed for user consumption, tend to show-up in error messages. Improving debugging for ZyTyG expressions, and for languages that support views in general, is an open issue.

The current strategy has the compiler type-check the underlying zygotes, which proves nothing about the safety of the DSEL expression they encode. An open issue is whether a host supporting a rich type semantics can relate the type-correctness of the encoding zygotes to the type-correctness of the DSEL expression.

5 Conclusion

The ZyTyG strategy shows how some simple syntactic properties in an object-oriented language permit defining a user-friendly DSEL. Many DSLs have been designed with such a syntax. Keeping the same syntax when the DSL is embedded means the learning curve can be reduced.

More generally, this paper shows how views can fruitfully be applied to transparently convert data and concepts between the DSEL and the host language. Views very naturally define the boundaries between two conceptual universes: a set of views is a complete definition of such a boundary. Both the scope of the boundary — through input and output types of views — and the crossing rules — trough the views' semantics — are defined.

Let me conclude by the following two observations. Firstly, a DSEL can transparently integrate with the host language without macro support. Secondly, DSEL designers that use an object-oriented host language have much to gain from using a language that supports views such as Scala.

References

[1] D. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming*, chapter 10–11. Addison-Wesley, 2005.

[2] D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: Tools for implementing domain-specific languages. In *Proceedings of the 5th International Conference on Software Reuse*, pages 143–153, 1998.

[3] M. Bravenboer and E. Visser. Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In *Proceedings of the 16th Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 365–383, 2004.

[4] K. Czarnecki, J. O'Donnell, J. Striegnitz, and W. Taha. DSL implementation in MetaOCaml, template Haskell, and C++. In *Proceedings of the International Seminar on Domain-Specific Program Generation*, pages 51–72, 2004.

[5] P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4es):196, 1996.

[6] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, 2005.

[7] M. Odersky. *The Scala Language Specification 2.0.* École polytechnique fédérale de Lausanne, Switzerland, March 2006.

[8] M. Odersky et al. An overview of the Scala programming language. Technical Report LAMP-REPORT-2006-001, École Polytechnique Fédérale de Lausanne, 2006.

[9] O. Shivers. A universal scripting framework, or Lambda: the ultimate "little language". In *Concurrency and Parallelism, Programming, Networking, and Security: Proceedings of the Second Asian Computing Science Conference*, pages 254–265, 1996.

[10] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages. Technical Report SEN-R0032, Centrum voor Wiskunde en Informatica, 2000.