Sectional Domain Specific Languages

Walter Cazzola
DICo, University of Milano, Italy
cazzola@dico.unimi.it

Ivan Speziale
DICo, University of Milano, Italy
ivan.speziale@studenti.unimi.it

ABSTRACT

Nowadays, many problems are solved by using a domain specific language (DSL), i.e., a programming language tailored to work on a particular application domain. Normally, a new DSL is designed and implemented from scratch requiring a long time-to-market due to implementation and testing issues. Whereas when the DSL simply extends another language it is realized as a source-to-source transformation or as an external library with limited flexibility.

The Hive framework is developed with the intent of overcoming these issues by providing a mechanism to compose different programming features together forming a new DSL, what we call a *sectional* DSL. The support (both at compiler and interpreter level) of each feature is separately described and easily composed with the others. This approach is quite flexible and permits to build up a new DSL from scratch or simplifying an existing language without penalties. Moreover, it has the desirable side-effect that each DSL can be extended at any time potentially also at run-time.

Categories and Subject Descriptors: D.3.3 [Programming Languages]: Language Constructs and Features; D.3.4 [Programming Languages]: Processors—*Compiler Generators*.

General Terms: Languages.

Keywords: DSL, Compilers, AOSD, Modularity.

1. INTRODUCTION

DSLs are used to solve several problems, such as typesetting documents and code (TEX/LATEX, lout, ...), to express and verify constraints in several domains (OCL, iLOG CP, C4J, ...) and to coordinate the computation and/or to data query (Linda, SQL, ...).

In some cases, these are simply a bunch of programming features — useless standalone — embedded in a general purpose programming language or provided as external libraries (e.g., Linda and SQL). More often, they are Turing complete programming languages devoted to a specific aim (e.g., LATEX). In both cases there are some issues that hamper their realization/usage: in the latter case, to implement and test a new DSL requires time and often it also has a steep learning curve; in the former case the learning curve is smoother but performances and flexibility are often compromised especially when the DSL is realized as program transformation towards another high-level programming language. Moreover, in general, it is quite hard to tailor an existing DSL to the needs of a given problem or to let coexist features coming from two or more DSL into a single programming language.

Let us consider to need the LATEX's typesetting capability accessible from our preferred scripting language to improve how matrices and vectors are typeset. Currently, we could follow two approaches:

1. to generate a complete LATEX program from the normal output, to pass it to the LATEX compiler, to get the result back and to visualize it as done by the `preview-latex`[1] component of the AUCTEX emacs plug-in;
2. to develop a new DSL integrating the desired feature in the original programming language, similarly to how plasTEX[2] and PyTEX[3] work.

The first approach results easier to use/learn and more transparent to the user that just need to know the original programming language but it is limited to what is provided, — e.g., AUCTEX previews only the formulas. On the other side, the second approach originates a new DSL that sounds odd to the experts of both languages without neglecting the time necessary to develop it.

In our view, the observed problems derive from the monolithic approach adopted to define a programming language and to implement its compiler. Classic approaches [1] to programming language definition and to compiler designing are *grammar-centric*; grammars are used to describe the syntax of a programming language and how a program written in that language should be translated by the compiler (*syntax directed translation*). Even if the compiling process can be modularized the language definition cannot, this, in our view, is the major obstacle to render a programming language easily extensible.

In this work, we present our approach, called Hive, to DSL development that exploits *parsing expression grammars* [4] to render the language definition sectional and extensible and aspect-oriented technology to support the sectional construction of a compiler by composing programming features from several languages.

2. HIVE

Our approach to compiler/interpreter building is inspired by Hyper/J's multi-dimensional separation of concerns and basically reflects the fact that the DSL has a sectional definition and each language feature can be easily plugged and unplugged. A complete compiler/interpreter built up with Hive is the result of a composi-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DSAL'09, March 3, 2009, Charlottesville, Virginia, USA.
Copyright 2009 ACM 978-1-60558-455-3/09/03 ...$5.00.

[1] `www.gnu.org/software/auctex/manual/preview-latex/`
[2] `plastex.sourceforge.net`
[3] `www.pytex.org`

tional process involving several building blocks. We call *modules* these basic units. Each module encapsulates a specific concern, such as the syntactical aspect of a loop, and thus is bound to a precise *role*, the syntax definition in our example. The role category determines where a module will be attached to. Roles that are a sort of *dimensions* in the Hyper/J parlance, are bound to the phases of the compilation and interpretation process.

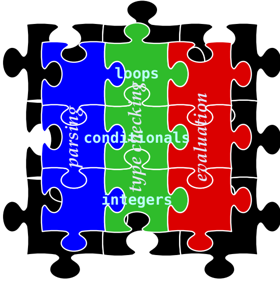

Figure 1: Sectional DSL

The full definition of a language construct is the collection of the modules addressing the roles pertaining to that construct; in Hive, we call *slices* this collection of modules. For instance, the slice regarding a loop may be composed of modules with a syntactical, a type checking and a evaluation role. In this scenario designing a domain specific language consists of defining a set of slices and composing them together. In Fig. 1 is depicted the general multi-dimensional structure of a DSL developed by using Hive, the colored jigsaw pieces are modules, those in the same row cope to describe the same feature and are part of the same slice whereas their color specific which role they play. Not necessarily the number of modules composing each slice is always the same nor a module for each role must be present in a slice.

To plug a slice in, we need a mechanism to precisely select the insertion point inside the compiler/interpreter. The process for selecting these insertion points, or joint points in aspect-oriented parlance, is naturally grammar driven: they correspond to the nonterminal symbols of the grammar; a grammar that dynamically grows as new slices are plugged in. In our case the code to be introduced at the join points, advice in aspect-oriented jargon, participates to define/implement the compiler/interpreter of the new DSL and consists of the grammar productions (in the syntactic module) with the related semantic action routines (in the other modules).

The Hive approach to compiler/interpreter building is *symmetric* [6], i.e., there is not a basic language to modify in order to get the new DSL rather it is generate by the slices composition. The composition specification defines the grammar join points and its advice. A complete compiler/interpreter reifies its grammar join points, so that it can be subsequently extended with new productions. A pleasant effect of symmetric composition is that many slices can be easily reusable by different domain specific languages. Beyond slices that have a syntactical dimension, there are some which do not have any. These slices encapsulate those concerns that could affect all the others crosswise, such as memory or symbol table management. We call these, *endemic slices*. Conceptually, we can imagine to project all the slices making up a compiler/interpreter into a bi-dimensional space (see Fig. 1). While slices pertaining to some constructs of the DSL are placed somewhere inside the bi-dimensional space, endemic slices slip out (the black jigsaw pieces in Fig. 1).

Our framework provides multi-dimensional separation of concerns applied to compilers/interpreters. The whole structure of the compiler/interpreter is the result of the composition of the slices, each regarding a particular construct of the final DSL. The compiler/interpreter can be orthogonally decomposed into a chain of phases/modules, such as parsing, type checking and so on. Building from scratch a new DSL consists of selectively reusing only those slices that best fit to our needs and composing them with the new ones.

2.1 Hive Vocabulary

Since we are switching DSL construction to an aspect-oriented approach, we need to show how the new terminology differs from the classic one.

As introduced before, in Hive, join points correspond to nonterminals and advice correspond to productions. In Hyper/J similar concepts are expressed during concerns mapping and hypermodules specification as names and features. In fact, while in the former we define which code pertains to which features, in the latter we decide how the code have to be composed.

The main difference between traditional aspects and what in Hive we call *module*, is that an aspect requires a base code to be woven into. Furthermore a module defines only its type, while the way it gets composed is provided in a later stage. Instead within an aspect the programmer has to specify the pointcut, which the advice is applied to. Actually in our framework we do not have a sophisticated language for expressing join points selection but we demand this topic for future works. In our context, a join point selection mechanism should allow a programmer to specify context driven compositions, such as how to change the meaning of an instruction depending on which scope it is executed, e.g. in a loop.

The main matching between Hive and Hyper/J is the *slice* concept. We consider a slice as the collection of those modules with different roles pertaining to the same construct. Although hyperslices allow the encapsulation of more generic features, they share other peculiarities such as declarative completeness and loosely coupling between slices.

Orthogonally to the slices we have the *dimension*; they are a collection of modules as well but characterized by the fact they play the same *role*; each role is a compiler/interpreter phase and the modules are contributing to the definition of the corresponding phase. To some extents, dimensions in Hive reassembles dimensions in Hyper/J.

3. FRAMEWORK IN DETAIL

The keys for implementing this approach to compiler/interpreter engineering are a flexible language for roles definition and a tool for integrating roles into slices and building up the compiler and/or interpreter.

3.1 Slices, Modules and Roles Definition

Each slice describing a certain programming feature is defined as a collection of modules pertaining to that feature; modules are defined through their role. The role concept plays the *glue* role in the slice definition.

```
slice for {
  module for with role syntax;
  module generic-loop with role type-checking, evaluation;
}
```

To separate the module from the slice definition will allow to have a composition mechanism with a fine granularity — a module can be used in the definition of several slices and modules from several slices can be (re)used to define another slice.

At the moment, the Hive framework supports only few dimensions: parsing (role `syntax`), type checking (role `type-checking`) and run-time evaluation (role `evaluation`); in the future more dimensions will be available.

Each dimension is processed by a specific back-end. Modules pertaining to the parsing dimension cope to form the DSL grammar and are processed by a parser generator; actually they are backed by Rats! [5]. Rats! allows a clean and modular grammar definition exploiting the power of *parsing expression grammars* (PEGs) [4].

PEGs are a recognition-based formalism, explicitly designed for describing machine oriented syntax. PEGs are close under composition, intersection and complement. PEGs can be parsed in linear time by the *packrat parser* [3]; a packrat parser is essentially a top-down parser with memoization capability. In broad terms, grammar manipulation can be brought back to handle methods. An unusual feature of PEGs is that they are scannerless, thus lexeme specification has to be done within the grammar. All those characteristics motivated our choice.

Inside each dimension it is possible to use some functionality provided by the corresponding back-end — e.g., the `syntax` role allows to use both regular and quoted expressions, such as the term `"for"` to identify the homonym keyword in the language. Moreover, data can pass from a dimension to another through *attribute*s associated to the grammar nonterminals (aka join points) — look at *attributed grammars* in [1] for the general idea; the attributes are defined by the colon (`:`) operator and accessed through the tilde (~) operator. The following piece of code shows two modules of the `for` slice: the parsing and evaluation dimension; these modules have four attributes (`init`, `condition`, `increment` and `statements`) that permits to the code-generator to generate the code for the source parsed by the parser.

```
module for {
  role(syntax) with assign-stmt, inc-stmt {
     statement ⇐
       "for" "("
          assign-stmt:init ";"
          bool-expression:condition ";"
          inc-stmt:increment
       ")" "{" statement-list:statements "}"
  }
  role(evaluation) {
    ~init ; while( ~condition ) { ~statements ; ~increment ; }
  }
}
```

A module can be derived from or can use other existing modules; this dependency is described by the **`with`** keyword.

The modules with a `syntax` role will form the syntax of the DSL when composed. They introduce new productions pivoting on some common nonterminals, such as `statement`, `stmt-list` or `bool-expression` or on nonterminals defined (and potentially exported to other slices) in the inherited slices, e.g., `assign-stmt` and `inc-stmt`. In the above example, the new production describes the `for` syntax in terms of `bool-expressions` and `statements` and pegs the new syntax to the `statement` nonterminal.

The modules with an `evaluation` role will give a semantics to the introduced slice, i.e., they teach to the interpreter how to evaluate the new syntax. The semantics is given as piece of code with some attributes (distinguishable by the ~ operator) that need to be expanded. Since our prototype works on the Java virtual machine the evaluation code is Java code but nothing prevents from defining the back-ends in different programming languages.

Endemic slices are somewhat different from normal slices: they do not introduce linguistic features rather they provide ancillary stuff crosscutting the implementation of many linguistic features. The symbol table and the memory management implementation are typical examples of endemic slices.

To use an endemic slice within a module, it is necessary to list it in the requirement (**`with`**) list of the module and to compose it with the other slices when we are building up the compiler/interpreter for the DSL.

Since an endemic slice embeds a crosscutting concern for several dimensions, — e.g., the symbol table can be accessed by the type-checking dimension and by the evaluation dimension — is unthinkable to limit its use to only a dimensions. Similarly to the evaluation code, the ancillary stuff introduced by an endemic slice is written in Java.

```
module less-than-expression {
  role(syntax) {
    bool-expression ⇐
      expression:lvalue "<" expression:rvalue
  }
  role(type-checking) with symbol-table {
    if( ! ~lvalue instanceof IntegerLiteral ) {
      Object o = getSymbol(~lvalue);
      if( ! o instanceof IntegerLiteral )
        throw new RuntimeException();
    }
    if( ! ~rvalue instanceof IntegerLiteral ) {
      Object o = getSymbol(~rvalue);
      if( ! o instanceof IntegerLiteral )
        throw new RuntimeException();
    }
  }
  role(evaluation) with symbol-table {
    Object first, second;
    ~result = Boolean.FALSE;
    if( ! ~lvalue instanceof IntegerLiteral )
      first = getSymbol(~lvalue);
    else first = ~lvalue;
    if( ! ~rvalue instanceof IntegerLiteral )
      second = getSymbol(~rvalue);
    else second = ~rvalue;
    if( first.compareTo(second) == -1 )
      ~result = Boolean.TRUE;
  }
}
```

The `less-than-expression` slice embeds the < predicate. It shows how the `evaluation` dimension get an explicit return value to be used in the evaluation of this expression by assigning a value to the `result` variable implicitly defined for each slice. In our example the value of the expression should be a boolean and it is set accordingly; its value can be accessed by calling the default method called `getResult()`. To achieve a correct composition is not mandatory to set `result`, in fact some constructs, such as `print` statements, do not need to any return value.

3.2 Composition phase

Composition is basically a two step process driven by a configuration file that permits to identify all the involved slices. The first composition step affects the composition between modules concerning the same slice, while the second regards slices integration. During modules composition only dependencies between modules of the same slice are solved. The composition between slices is responsible for defining the final grammatical structure.

The whole composition process pivots on the grammar nonterminal concept or if you prefer on what we call *grammatical join points*. Some *remarkable nonterminal*s — such as `declaration`, `expression`, `statement` and so on — are given (as empty sets) and can be used to build up the language from the basics. New nonterminals can be easily added by definition, as happens in

```
assign-stmt ⇐ identifier "=" expression
```

where `assign-stmt` is added to the remarkable nonterminals and can be used in the definition of new slices.

To obtain the desired language structure, it is necessary to enroll all the necessary remarkable nonterminals by listing them in the composition file. Moreover we have to list all the slices that will compose the final DSL; from this list the composition process can

determine the whole bunch of nonterminals and which is the grammar structure for the DSL. Similarly it can build up the type system and the interpreter.

```
language for-dsl {
  nonterminal: bool-expression, stmt-list, statement
  slices:
    for, assignment, print, symbol-table, increment,
    identifier, less-than-expression, integer-literal
}
```

In the composition file all the pending symbols must be linked, to obtain the desired grammatical structure. The skeleton is introduced into the composition specification by means of a **`nonterminal`** declaration. Those joint points do not provide any hierarchy, the association among nonterminals and join points (when it is no the obvious one) is up to the DSL creator. The key concept towards language extension and reuse is that both initial join points and slices are reified into the resulting compiler/interpreter. For that reason is possible to augment the DSL by adding some new construct, modifying or deleting the old ones.

3.3 Reusability and extensibility

To complete the overview of the Hive approach and to give a glimpse to its merits, we examine the extensibility and reusability of a DSL developed in Hive.

In the previous pages, we have defined, what we could call the *for-language*, a quite trivial DSL with just the sequence, the for loop (with assignment and auto increment operations), variables (without declaration), literals and the < operator between expressions. At the moment, integer is the only supported type but it is quite desirable to have a more complex type system.

Let us suppose to need a *for-language* with integers and characters, as the one described by the following grammar (some trivial productions are omitted for sake of space).

```
StmtList    ::=  Stmt ; StmtList | Stmt ;
Stmt        ::=  for ( AssignStmt ; BoolExpr ; IncStmt ) { StmtList } |
                 AssignStmt | print Id | IncStmt
IncStmt     ::=  Id ++
AssignStmt  ::=  Id = Expr
Expr        ::=  BoolExpr | Id | IntLiteral | CharLiteral
BoolExpr    ::=  Expr < Expr
```

Since the *apparent* differences from the *for-language* are minimal should be easy to get its compiler/interpreter from those already realized. Actually, such a kind of extension affects in several way a programming language and its compiler/interpreter:

- new literals (the characters) are available;
- the existing constructs must be adapted to support the new type, e.g., the < operator must compare characters too;
- the type checking phase needs to check if two expressions are type-compatible and takes some actions (coercion, promotion, to raise an exception, . . .) if not.

As stressed by Bracha [2], to turn on/off or extend a type system are desirable features but not so easy to realize since they affect the whole structure of the compiler/interpreter as a pandemic.

Hive thanks to its sectional structure permits to easily replace the type-system of the whole DSL. To ease the process we introduce a new endemic slice encapsulating the type-checking policy.

```
slice type-manager {
  role(endemic) {
    public Hashtable<Class,Class> CompatibilityTable;
    public checkCompatibility(Class, Class) { ... }
  }
}
```

How the type checking is carried out is out of the scope of this paper; let us simply consider that the `type-manager` slice provide the other slices with a `checkCompatibility()` method.

Given that to adapt the existing `less-than` slice means simply to define a new module embedding the type checking for the < operator which exploits the `type-manager` slice.

```
module less-than-polymorphic {
  role(type-checking) with type-manager {
    if (!checkCompatibility(~lvalue, ~rvalue))
      throw new TypeException("Incomparable Types!!!");
  }
}
```

The slice definition for the new polymorphic < operator must be composed of the modules with the roles `syntax` and `evaluation` previously defined and the module with role `type-checking` just defined and the initialization of the `type-manager` endemic slice.

```
slice less-than {
  module less-than-expression with role syntax, evaluation;
  module less-than-polymorphic with role type-checking;
  init {
    CompatibilityTable = new Hashtable<Class, Class>();
    CompatibilityTable.put(int.class, int.class);
    CompatibilityTable.put(char.class, char.class);
  }
}
```

From the given initialization integers and characters are not comparable elements. Similarly we can easily extends all the available slices (if necessary) getting the desired DSL.

4. CONCLUSIONS

This work present our initial work on designing (and realizing) a development framework (a programming language and compiler/interpreter generators), named Hive, to support flexible and extensible DSL design and implementation. The approach has an aspect-oriented architecture considering the implementation of a programming feature as a crosscutting concern at compiler/interpreter level. The proposed approach has the benefit to permit to easily define new DSLs as a variant of existing programming languages by removing or adding slices (our modularity unit for a programming feature/crosscutting concern) from/to the existing compiler/interpreter. In the future, we intend to deeply work on the slice definition and composition mechanism and to their impact in software evolution and maintenance merging the sectional-compiler idea with just-in-time compiler technology.

5. REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986.

[2] G. Bracha. Pluggable Type Systems. In *Proc. of Workshop on Revival of Dynamic Languages*, Oct. 2004.

[3] B. Ford. Packrat Parsing: Simple, Powerful, Lazy, Linear Time. In *Proc. of ICFP'02*, Pittsburgh, USA, Oct. 2002.

[4] B. Ford. Parsing Expression Grammars: a Recognition-Based Syntactic Foundation. *Proc. of the POPL*, pp. 111-122. 2004.

[5] R. Grimm. Better Extensibility through Modular Syntax. In *Proc. of PLDI'06*, pp. 38–51, Ottawa, Canada, June 2006.

[6] W. Harrison, H. Ossher, and P. Tarr. Asymmetrically vs. Symmetrically Organized Paradigms for Software Composition. Technical Report RC22685 (W0212-147), IBM, Dec. 2002.