Cedalion – A Language Oriented Programming Language (Extended Abstract)*

David H. Lorenz Boaz Rosenan

The Open University of Israel

Abstract

Implementations of language oriented programming (LOP) are typically either language workbenches, which facilitate the development of external domain specific languages (DSLs) with projectional editors, or host languages for internal DSLs that are parsed. In this work, we present Cedalio—a novel approach to LOP, along with a prototyped programming language and workbench implementing our approach, which uses internal DSLs in conjunction with projectional editing. To the best of our knowledge, Cedalion is the first language workbench to implemented such an approach.

1 Introduction

Language Oriented Programming (LOP) is a paradigm that puts the programming language at the center of the software development process. LOP focuses on the use of *domain specific languages (DSLs)*. The LOP software development process consists of three parts: a definition of a DSL or several interoperable DSLs; the implementation of these DSLs by means of interpreters, translators or compilers; and the development of the software using these DSLs [11, 1, 2].

LOP has several advantages, directly derived from use of DSLs: (1) *Separation of concerns:* The declarative design (handled by the DSL code) is decoupled from the imperative implementation (handled by the DSL implementation) of the application. (2) *Maintainability:* Programming in DSLs reduce the application code size. (3) *Portability:* Targeting a new architecture can be done by re-implementing the DSL only, without modifying the application code. (4) *Reusability:* The language implementation can be used for other applications within the same domain.

However, these advantages come at a cost: the cost of defining and implementing the DSLs; the cost of training developers to use them; the cost of developing language-specific tools to support development (or alternatively, the loss of productivity in their absence); and the loss of runtime performance in comparison with equivalent code written and fine-tuned in a lower-level general-purpose language.

This cost depends on how DSLs are implemented. LOP is typically realized by using either internal DSLs or external DSLs. *Internal DSLs* (also called *embedded DSLs* [6]) are DSLs implemented as a set of definitions in a general-purpose programming language, called the *host language*, e.g., LISP (and its dialects), Smalltalk, Haskell, Ruby, etc. *External DSLs* are DSLs implemented using interpreters, translators or compilers, written in some other language, and are thus external to the language in which they are defined.

The cost of developing external DSLs often renders LOP impractical, unless supported by Language Workbenches. *Language Workbenches* are integrated development environments (IDEs) designed to facilitate the LOP process [2], focusing on external DSLs. They provide tools for designing DSLs, implementing them (as compilers, translators or interpreters) and developing applications using these DSLs. Language workbenches ease the task of defining a DSL, and support the creation of dedicated editors, often with rich tooling for that DSL, at very little cost. This helps reduce the cost

*Research supported in part by the *Israel Science Foundation (ISF)* under grant No. 926/08.

of using LOP, thus making it a viable alternative to conventional programming paradigms.

Existing implementations of language workbenches for LOP use projectional editors to facilitate the development of external DSLs. *Projectional editors* are editors that modify the *model* through a *projection* to a *view* [3]. Projectional editors allow modifications to the model which are visualized as changes to the view. Notable language workbenches include *Jetbrains' MPS* [1], the *Intentional Software Workbench* [9] and *MetaEdit+* [10].

Contribution In this work, we present a novel approach to LOP that uses *projectional* editors to facilitate the development of *internal* DSLs. To the best of our knowledge, no work has yet been reported on projectional editing for internal DSLs. The closest work we know of is ModelTalk [5, 4], which provides a language workbench with rich tooling for internal DSLs. However, ModelTalk does not provide projectional editing. Rather, the model is edited directly as text.

Internal DSLs reuse the host language's interpreter or compiler. This makes their implementation concise, letting the DSL developer focus mainly on the semantics of the DSL. All DSLs internal to a certain host language enjoy symbolic integration [2] to the host language, i.e., the ability to used symbols of the host language in the DSL and vice versa. Unlike external DSLs, where the entire DSL implementation must rely on other ("prior" or pre-existing) languages, with internal DSLs constructs can be defined incrementally, one on top of the other, and even recursively, defining a certain construct in terms of itself. This makes the DSL implementation easier to build and maintain.

On the other hand, external DSLs enjoy full freedom in the definition of their syntax and semantics (limited only by computability and complexity), in contrast to internal DSLs, which are limited by the syntax and semantics of the host language. A good host language allows very little restrictions on the DSL semantics (i.e. provide good semantic extensibility), but the syntactic restrictions are unavoidable, due to the limitations of parsing.

DSLs using projectional editing have the advantage of not being bound by the limitations of parsing. Even if the language is mostly textual (as with MPS or Intentional Software), its syntax does not have to meet restrictions paused by practical parsing algorithms such as LALR or LL(k). This is extremely important when using several DSLs in a single source file. Even if every DSL is by itself within the desired class (e.g. LALR or LL(k)), there is no way to guarantee that the fusion of the grammars required to parse the combined source file will still be in that class, or even unambiguous. On the other hand, when using projectional editing, parsing does not take place and grammar classes are irrelevant. Inambiguity is guaranteed by the model, using unique node types to depict different constructs in different languages. Even if two constructs happen to have similar projections, they are still different deep down. In addition, even for a DSL that is mostly textual (such as one developed in MPS), the syntax is not limited to a stream of characters. Colors, symbols, font styles and sizes can be used as part of the projected syntax, thus improving expressiveness.

$$\begin{cases} S ::= aSb \\ S ::= \varepsilon \end{cases}$$

Figure 1: A simple grammar

Outline In section 2 we will demonstrate how easily a DSL can be defined over a host language such as Prolog. In section 3 we will introduce Cedalion, our programming language and workbench built around it. In section 4 we will discuss Cedalion and compare it to Prolog, and in section 5 we will draw our conclusions.

2 Motivating Example

To get a sense of the ease of defining internal DSLs, we provide a short example DSL example. We use Prolog as the host language. Prolog is not well known as a potential host for internal DSLs, but it is well suited for the job [7]. In our example, we shall define a DSL for implementing recursive-descent parsers based on BNF grammars. For illustrative purposes and as a test case for our DSL, we consider the simple language defined by the grammar in Figure 1.

DSLs are designed to provide the highest possible level of abstraction when approaching a problem. For this reason we wish our DSL code to look as similar as possible to its specifications (Figure 1). Figure 2 provides such code, using Prolog's syntax.

```
s::=[a], s, [b].
s::=[].
```

Figure 2: Parser implementation using the DSL

Now all that is left to do is to make these specifications executable. Figure 3 shows how this can be done.

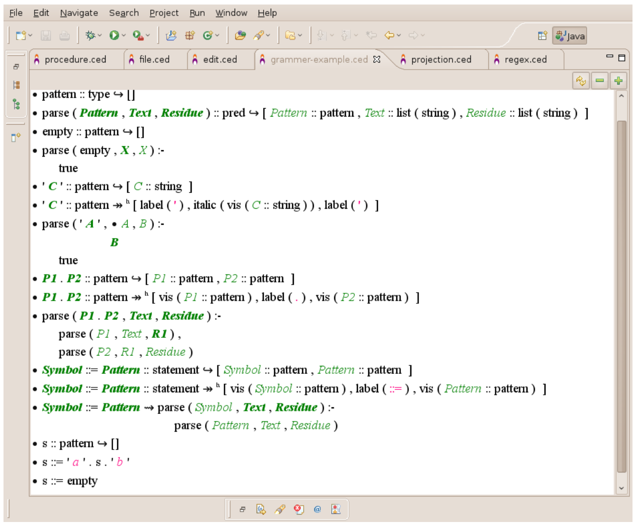
```
:- op(1100, xfx, '::=').
parse([], X, X).
parse([T], [T|L], L).
parse((P1,P2), X, Y) :-
        parse(P1, X, X1),
        parse(P2, X1, Y).
parse(P, X, Y) :-
        (P ::= Q),
        parse(Q, X, Y).
```

Figure 3: BNF DSL Definition

The first line defines the BNF ::= operator *syntactically*, allowing the Prolog parser to parse the code in Figure 2. It is followed by four clauses defining the *parse* predicate, each defining a single construct in our language. The first clause defines an empty list, depicting empty input. The second defines a list with one element, depicting a terminal. It is followed by a definition of the comma operator, depicting concatenation, followed by a semantic definition of the ::= operator. This last definition is not a definition of the ::= predicate in the Prolog sense. It is a definition of its *meaning* made by using it (the definition of the predicate itself is given in Figure 2). It could be read:

$$P{::=}Q \rightsquigarrow parse(P,X,Y){:\text{-}}parse(Q,X,Y)$$

meaning that every clause of the form $P{::=}Q$ can be replaced by a clause of the form $parse(P,X,Y){:\text{-}}parse(Q,X,Y)$. With both the DSL definition and code in place, the goal $parse(s,[a,a,b,b],[])$ will succeed, while the goal $parse(s,[a,a,a,b,b],[])$ will fail.



```
• pattern :: type ↪ []
• parse ( Pattern , Text , Residue ) :: pred ↪ [ Pattern :: pattern , Text :: list ( string ) , Residue :: list ( string ) ]
• empty :: pattern ↪ []
• parse ( empty , X , X ) :-
     true
• ' C ' :: pattern ↪ [ C :: string ]
• ' C ' :: pattern ↠ h [ label ( ' ) , italic ( vis ( C :: string ) ) , label ( ' ) ]
• parse ( ' A ' , • A , B ) :-
              B
     true
• P1 . P2 :: pattern ↪ [ P1 :: pattern , P2 :: pattern ]
• P1 . P2 :: pattern ↠ h [ vis ( P1 :: pattern ) , label ( . ) , vis ( P2 :: pattern ) ]
• parse ( P1 . P2 , Text , Residue ) :-
     parse ( P1 , Text , R1 ) ,
     parse ( P2 , R1 , Residue )
• Symbol ::= Pattern :: statement ↪ [ Symbol :: pattern , Pattern :: pattern ]
• Symbol ::= Pattern :: statement ↠ h [ vis ( Symbol :: pattern ) , label ( ::= ) , vis ( Pattern :: pattern ) ]
• Symbol ::= Pattern ⇝ parse ( Symbol , Text , Residue ) :-
                     parse ( Pattern , Text , Residue )
• s :: pattern ↪ []
• s ::= ' a ' . s . ' b '
• s ::= empty
```

Figure 4: A Screenshot of the Cedalion Workbench

3 LOP with Cedalion

We introduce *Cedalion*, a logic programming language, designed specifically as an LOP platform for developing and using internal DSLs in conjunction with projectional editing. The choice of logic programming was done due to its clean declarative nature, with features such as pattern matching and backtracking making it highly expressive. However, similar results can be achieved by using projectional editing in conjunction with other programming paradigms.

We implemented an IDE for Cedalion development as an Eclipse plug-in. The Cedalion Workbench acts as the projectional editor for Cedalion code. Figure 4 shows a screenshot of the workbench, showing an implementation of a DSL similar to the one defined in Figure 3. Note that this code cannot be edited directly as text.

The definition is longer then the one in Prolog, as it defines a signature for all new constructs (using the $\hookrightarrow$ operator), and defines *projections* for some of them (using the $\twoheadrightarrow$ operator). The three last lines at the bottom of the screenshot are the implementation of the grammar in Figure 1 using the DSL defined above it. Apart from the two production rules, it also contains a signature definition for *s*, defining it as a *pattern*. Note that placing DSL code in the same source file as a the DSL definition is usually only possible for internal DSLs.

Cedalion is statically typed. It uses a Hindley-Milner [8] type inference, adjusted for logic programming in support for internal DSLs. Type signatures need to be defined explicitly for each concept, but type inference helps the user add them, often with a single click. The type system enforces the DSL schema and guides the user in writing correct DSL code. For example, hovering the mouse over logic variables while editing the code in Figure 4 will emit a tooltip depicting the variable's type, as inferred by the type system. When the system detects a type violation, a red rectangle with an error icon at its left is placed around the location of the error, and a tooltip depicts the nature of the error. Double clicking the icon displays a list of possible fixes (if any). Double clicking a suggested fix modifies the code and hopefully fixes the problem.

Cedalion allows for DSLs to be defined layer by layer. It allows the definition of higher levels of abstraction to "stand on the shoulders" of all the abstractions made until that point, hence its name. Due to space limitations, we do not include more screenshots here.

4 Discussion

Implementing projectional editing on top of Cedalion rather than directly on top of the Prolog semantics has two major benefits. The first benefit is the fact Cedalion is statically type whereas Prolog is dynamically typed. This means that there is no notion in Prolog of a *schema* for the DSLs, and no way to statically validate the DSL code, or to guide the developer in writing correct DSL code. Misspelled or misplaced DSL construct would result in unexpected behavior of the program (typically failure of a goal where success is expected), and finding the reason will typically require debugging. Cedalion's type system provides such enforcement statically.

The second benefit of Cedalion is its declarative versus Prolog's imperative behavior. Prolog allows *cuts* and *side-effects*. Cedalion does not, making it a pure declarative language.

Cedalion also has *rewrite-rules*, statements of the form $S_1 \rightsquigarrow S_2$, which are a special construct made to depict language extension. They are comparable with Horn clauses (statements of the form H:-B), but are more restricted. In the example in figure 4 we used a rewrite rule similar to the pseudo-code at the end of section 2.

5 Conclusions

In this paper, we described Cedalion, a programming language designed as a platform for LOP. We show how easily DSLs can be defined in Cedalion and how expressive they can be. Cedalion is a programming language and workbench designed to support LOP. As such, it is designed to serve as a host for internal DSLs. The choice of projectional editing in Cedalion avoids the limitations of parsed languages, especially when using several DSLs together. To our knowledge, Cedalion is the first to illustrate a programming language that can host internal *projected* DSLs. For illustrative purposes, we chose logic programming as the host paradigm, on top of which Cedalion can define DSLs. But the approach is applicable to other programming paradigms.

References

[1] S. Dmitriev. Language oriented programming: The next programming paradigm. *JetBrains onBoard*, 1(2), 2005.

[2] M. Fowler. Language workbenches: The killer-app for domain specific languages. *Accessed online from: http://www. martinfowler. com/articles/languageWorkbench. html*, 2005.

[3] M. Fowler. MF Bliki: ProjectionalEditing. *Accessed online from: http://martinfowler. com/bliki/ProjectionalEditing. html*, 2008.

[4] Atzmon Hen-Tov, David H. Lorenz, and Lior Schachter. ModelTalk: A framework for developing domain specific executable models. In *the 8^{th} OOPSLA Workshop on Domain-Specific Modeling*, pages 45–51, Nashville, TN, October 19-20 2008. *CoRR*, abs/0906.3423:(2009).

[5] Atzmon Hen-Tov, David H. Lorenz, and Lior Schachter. ModelTalk: A framework for developing domain specific executable models. *CoRR*, abs/0906.3423, 2009.

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

[7] T. Menzies. DSLs: A Logical Approach. *Accessed online from: http://courses. ece. ubc. ca/571f/index. html*, 2001.

[8] R. Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.

[9] C. Simonyi, M. Christerson, and S. Clifford. Intentional software. *ACM SIGPLAN Notices*, 41(10):464, 2006.

[10] J.P. Tolvanen and S. Kelly. MetaEdit+: defining and using integrated domain-specific modeling languages. In *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 819–820. ACM, 2009.

[11] M.P. Ward. Language-oriented programming. *Software-Concepts and Tools*, 15(4):147–161, 1994.