Designing Language-Oriented Programming Languages

Boaz Rosenan

Dept. of Mathematics and Computer Science
Open University of Israel
brosenan@cslab.openu.ac.il

Abstract

Today, language-oriented programming (LOP) is realized by using either language workbenches or internal DSLs, each with their own advantages and disadvantages. In this work, we design a host language for DSLs with language workbench features, thereby combining the two approaches and enjoying the best of both worlds.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Specialized application languages

General Terms Languages, Design

1. Introduction

Language-oriented programming (LOP) is a software development paradigm that places *domain-specific languages* (DSLs) at the center of the software development process [7, 1, 2]. With LOP, software is developed "from the middle out" [7], starting with a definition of a DSL, or several interoperable DSLs, going "up," implementing the software product using these DSLs, and going "down," implementing the DSLs themselves, and thus making the software executable.

DSLs play a key role in the realization of LOP, and consequently code developed using LOP is declarative, concise and close to the specification it is based on. DSLs are highly expressive, each DSL fitted to the problem domain it is designed to capture. DSLs increase reusability because the same DSL can be used to implement similar but different software products (e.g., different products in a software product line). Reusability is maximized when using several, interoperable DSLs. This increases granularity when sharing DSLs between software products, and thus improves reusability.

SPLASH'10, October 17–21, 2010, Reno/Tahoe, Nevada, USA.
ACM 978-1-4503-0240-1/10/10.

Fowler [2] makes a distinction between two kinds of DSLs: *external* and *internal*. *External DSLs* are DSLs that are implemented as compilers, translators or interpreters, and are thus *external* to the programming language in which they are implemented. *Internal DSLs* (also known as *embedded DSLs* [3]) are implemented using definitions in a pre-existing, usually general-purpose programming language, named the *host language*, and are thus internal to the host language.

These two kinds of DSLs have significant trade-offs. On the one hand, external DSLs provide their designers with full freedom in defining the desired syntax and semantics, while internal DSLs are bound to the syntax and semantics of the host language. On the other hand, internal DSLs reuse the compiler or interpreter of the host language, making their implementation much simpler in comparison with external DSLs. Internal DSLs are also better suited for DSL interoperability, since code in different DSLs written over the same host language is actually code in the host language.

Intuitively, external DSLs are better for going "up" in the LOP process, as they are potentially more expressive, while internal DSLs are better for going "down", as they are easier to implement. To make LOP an effective and practical paradigm, there is a need to balance the trade-offs between internal and external DSLs.

2. State of the Art

The limitations of both internal and external DSLs have biased the software industry in favor of conventional programming paradigms over LOP. Recent work attempts to remedy this by bridging the gap between internal and external DSLs through the use of language workbenches. *Language Workbenches* are Integrated Development Environments (IDEs) for defining, implementing and using external DSLs [2]. They address some of the limitations of external DSLs, allowing them to enjoy some of the features traditionally associated with internal DSLs. These include relieving the developer of the need to provide a parser for the DSL and supporting symbolic integration between DSLs [2]. They ease the definition and implementation of external DSLs by applying LOP to these tasks, i.e., by providing *meta-DSLs* for defining and implementing DSLs. A unique feature of language

workbenches is the fact they use projectional editing [2] as an alternative to parsing. With this, they edit the DSL abstract syntax tree (AST) directly by projection to a view, where the AST is considered a *model*. DSL interoperability is thus possible through interoperability between models. However, DSL implementation is still hard, and suffers from the limitations of code generation or those of interpretation (whichever is selected as the implementation method). The most notable language workbenches include MPS [1] and the Intentional Domain Workbench [6].

In contrast to language workbenches, our work takes the opposite direction, making internal DSLs enjoy features traditionally associated with external DSLs.

3. Approach

In this work we design a *host language* for *internal DSLs*, with properties currently associated mostly with language workbenches. These properties are: (1) the ability to define and use projectional editing, and (2) the ability to define and enforce *DSL schemas*, i.e., the set of rules defining valid DSL code. We call such a host language a *language-oriented programming language* (LOPL), as it is well suited to support LOP, just like object-oriented programming languages (OOPL) are suited to support OOP.

An LOPL can be based on the semantics of a non-LOP programming language, as long as that language meets certain criteria. First, it should be a good *host language* for internal DSLs. To allow projectional editing it needs to be able to reason about its own code, meaning that the language should to be *homoiconic* or *reflective*, at least to some extent. For projectional editing to be effective, the language should to be *minimalistic*, i.e., have a small number of node types in the AST representation of the code. This is important because projectional editing transforms this AST into visuals. Projection definitions can refer to a small number of node types, such as lists in Lisp, compound terms in Prolog or message-sends in Smalltalk. Projectional editing will be most effective if this small number of types comprises most of the AST. Since the code providing the projection definition needs to run at "design time," i.e., as the DSL code is being edited, *dynamic semantics* makes sense for LOPLs.

Static typing of the host language can help define and enforce DSL schemas. When used with projectional editing, static typing can make the editor "smarter," guiding the user to write valid DSL code to begin with. When using a dynamic host language, the type system can be implemented by using reflection.

4. Validation

To validate our approach, we developed *Cedalion* [4], an LOP language. Cedalion uses logic-programming as its core semantics, taking advantage of the homoiconic, minimalistic, and dynamic nature of Prolog. Another consideration for the selection of logic-programming is the ease of defining operational semantics for DSLs, in a clean and formal way.

Cedalion code is edited using a projectional editor, and the visualization of language constructs is customized by adding clauses to a predicate. Cedalion is statically typed, but its type system is implemented from within the language, as a set of predicates, rather then as part of the language. The predicates comprising the type system are activated from within the visualization mechanism, highlighting erroneous code and suggesting solutions.

LOP is realized in Cedalion by defining, implementing and using internal DSLs, using Cedalion as the host language. A DSL definition includes type signatures and projection definitions for all new constructs, while the implementation usually consists of deductions providing operational semantics for these constructs, as demonstrated by Menzies [5].

5. Conclusion

The main contribution of our work is in presenting a novel approach for bridging the gap between internal and external DSLs. Its uniqueness is in the direction we take: using internal DSLs, taking advantage of the ease of implementing them, while adding language-workbench features (namely projectional editing and enforcing a schema) to provide the freedom and safety in defining DSLs, qualities traditionally associated with external DSLs.

As a paradigm, LOP has a great potential in improving the way we write and maintain software. However, it is not widely adopted due to the limitations of DSLs, internal or external. LOP languages have the potential of changing that, with languages such as Cedalion paving the way.

References

[1] S. Dmitriev. Language oriented programming: The next programming paradigm. *JetBrains onBoard*, 1(2), 2004.

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

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

[4] D. H. Lorenz and B. Rosenan. Cedalion: A language oriented programming language. In *IBM Programming Languages and Development Environments Seminar*, Haifa, Israel, Apr. 14 2010. IBM Research - Haifa.

[5] T. Menzies. DSLs: A logical approach, 2001. Lecture Notes, EECE 571F, `http://courses.ece.ubc.ca/571f/lectures.html`.

[6] C. Simonyi, M. Christerson, and S. Clifford. Intentional software. *ACM SIGPLAN Notices*, 41(10):451–464, 2006.

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