A Web Application Is a Domain-Specific Language *

David H. Lorenz[1,2] †
[1]Open University, Raanana 43107, Israel
[2]Technion–Israel Institute of Technology, Haifa 32000, Israel
dhlorenz@cs.technion.ac.il

Boaz Rosenan[1,3]
[3]University of Haifa, Mount Carmel, Haifa 31905, Israel
brosenan@gmail.com

Abstract

We introduce a correspondence between the design space of web applications and that of domain-specific languages (DSLs). We note that while most web applications today are implemented in ways that correspond to external DSLs, very little attention is given to implementation techniques corresponding to internal DSLs. We contribute a technique based on internal DSLs, and demonstrate a web application implemented with our technique.

Categories and Subject Descriptors D.2.11 [*Software Engineering*]: Software Architectures—Domain-specific architectures

General Terms Language, Design.

Keywords Domain-specific language (DSL), Web application, *Von Neumann architecture*.

1. Introduction

Web applications and *Domain-Specific Languages (DSLs)* have many things in common. Both are built around some *schema* (a database schema and an abstract syntax), both give this schema a concrete, user-facing, *representation* (user interface and concrete syntax), and both give *meaning* to data structured under that schema (business logic and semantics, respectively). For both, the schema is domain-specific, describing entities from a single problem domain.

We formalize this observation in what we call the *App-DSL correspondence*. According to this correspondence, a web application corresponds to a DSL. An instantiation of the application's schema – a snapshot of the application's state at a given time – corresponds to an instantiation of the abstract syntax of a DSL – a single program. The evolution of the state of an application is similar to a DSL program being edited. In this case, the application's user interface acts as the editor, and in that it is inseparable from the concrete syntax. This is similar to some *Projectional Editing* (*PE*) [1] frameworks, where the concrete syntax of a language (DSL) is defined as a dedicated editor for that language.

A correspondence, such as the App-DSL correspondence, is useful if it allows us to transform methods known to be effective in one domain into novel methods in the corresponding domain, and apply them effectively in that domain. In this work we note that the state of the art techniques for implementing web applications typically correspond to the implementation technique of developing *external DSLs*, while the technique of developing *internal DSLs* (also known as *DSL embedding*) is not used in the context of web applications. To gain the advantages of DSL embedding in the context of web applications we identify the essential components for *application embedding*, and provide preliminary results in implementing a simple web application using this method.

2. The App-DSL Correspondence

The App-DSL correspondence starts with the realization that a database schema of an application and the abstract syntax of a DSL are both a form of type declaration. In the case of an application, the value defined by this declaration is the content of the database at a certain point in time, which constitutes the state of the application at that point. In the case of a DSL, a value of the type declared as the abstract syntax is a DSL program. Both the state and the DSL program are given human-readable and writable form. In the case of a DSL, this is its concrete syntax, while in the case of an application, this is its user interface. Both the DSL program and the state of the application are given meaning. In the case of a DSL, the meaning of the program is determined by the DSL's semantics. In the case of a web application, the meaning of the state is given by the business

* This research was supported in part by the Israel Science Foundation (ISF) under grant No. 1440/14.

† Work done in part while visiting the Faculty of Computer Science, Technion—Israel Institute of Technology.

SPLASH Companion'16, October 30 – November 4, 2016, Amsterdam, Netherlands
ACM. 978-1-4503-4437-1/16/10...$15.00
http://dx.doi.org/10.1145/2984043.2989220

logic, which determines how state changes and how queries are answered, based on the current state.

2.1 DSL Embedding

There are two distinct ways to implement a DSL. External DSLs are implemented in the form of compilers or interpreters, written in another programming language. Internal DSLs [2], on the other hand, are extensions to an existing host language, which are written in that same host language, and are therefore internal to it. Several different techniques exist for implementing such internal DSLs, corresponding to different kinds of host languages. These techniques are generally referred to as *DSL embedding* [3].

Comparing the state of the art in web application development to DSLs, it seems that the vast majority of existing techniques correspond to external DSL implementation. Although web applications correspond to languages, they are not treated as such when implemented. Web applications have been implemented in almost any programming language, but all resulting applications are external to the implementation language (they do not extend the implementation language and instead create something new).

There are significant trade-offs between internal and external DSLs. External DSLs give greater control over syntax and semantics compared to internal DSLs, at the cost of greater implementation effort. Internal DSLs are easier to implement as they reuse the host language implementation as well as the tools built around it, but that comes at the cost of needing to conform to the host language's restrictions. By examining what application development techniques correspond to DSL embedding, we discover similar trade-offs for web applications.

2.2 Application Embedding

DSL embedding is based on the concept of a host language. The App-DSL correspondence correlates a language to an application, meaning that a host language corresponds to a host application. This is an application that needs to be general-purpose – be able to support a wide variety of web applications. In DSL embedding, the DSL is given as a piece of program (e.g., a software library or a module). The App-DSL correspondence correlates programs to data, meaning that the specific application's implementation is represented as data in the host application's database. This data resides side-by-side with user data, corresponding to the DSL program, which in the case of internal DSLs resides side-by-side with the DSL implementation within the same host language program.

3. Evaluation

To show feasibility and to demonstrate the usefulness of our approach, we implemented a proof-of-concept host application named FishTank. This host application has the typical structure of a web application, leveraging a NoSQL database and the Node.JS platform. However, it is designed as a general-purpose host application, being able to store arbitrary data and apply arbitrary business logic, based on rules given to it by application developers. Our proof-of-concept implementation uses the CloudLog [4] data language for representing both the facts (user data) and rules to be applied over them.

To demonstrate FishTank's ability to host applications we implemented a small, though nontrivial, micro-blogging (Twitter-like) application in CloudLog, and loaded it onto FishTank. While the latency of the resulting application is significantly higher than what is achievable using standard methods, it gives reasonable usability. As FishTank was designed for horizontal scaling, our micro-blogging application inherits this property out of the box. Since CloudLog is a purely-declarative language, the micro-blogging application we implemented is completely declarative, with the only imperative pieces of code being a few short lines describing what happens when a user clicks a button in the user interface.

4. Conclusion

The App-DSL correspondence illuminates the fact that web applications are commonly developed in a way that corresponds to external DSLs, while the equivalent of internal DSLs is unexplored territory. Our preliminary results show reasonable usability and applicability, with many opportunities for improvement and growth. The main benefit that can be seen in our proof-of-concept is the ease in which an application can be implemented, in an almost entirely declarative manner. Defining business logic declaratively is far from trivial, since it involves the manipulation of state. By treating state as a program we turn the state manipulation problem into a problem similar to projectional editing. The latter is easier to solve declaratively. It allows a separation of concerns between the imperative host application and the declarative concrete application.

References

[1] M. Fowler. Projectional editing. Martin Fowler's Bliki. `http://martinfowler.com/bliki/Projectional-Editing.htmlx`.

[2] M. Fowler. Language workbenches: The killer-app for domain specific languages, 2005.

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

[4] D. H. Lorenz and B. Rosenan. Separation of powers in the cloud: Where applications and users become peers. In *Proceedings of the 2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2015)*, pages 146–159, Pittsburgh, PA, USA, Oct. 2015. ACM.