Building Domain Specific Languages and Polyglot Applications with GraalVM

Presented to the Faculty of Computer Science and Mathematics
University of Applied Sciences Regensburg
Study Programme:
Master Computer Science

Master Thesis

In Partial Fulfillment of the Requirements for the Degree of
Master of Science (M.Sc.)

Presented by: Christian Paling
Student Number: 123456

Primary Supervising Professor: Prof. Dr. Michael Bulenda
Secondary Supervising Professor: ??

Submission Date: ??

Thesis Declaration

Abstract

Table of Contents

1 Introduction

Business as usual

2 Domain Specific Languages

Before diving into the technical details and the implementation of DSLs on top of GraalVM, some background information is necessary to lay a foundation for the upcoming chapters of this thesis. First, the term *domain specific language* is properly defined and a distinction between different types of DSLs is made. Afterwards, benefits as well as problems of DSLs and the usage thereof are discussed. Lastly, basic techniques and examples of how to implement the different types of DSLs are introduced. The contents of this chapter are based heavily on *Domain-Specific Languages* by *Martin Fowler* [1] which can be consulted for further information about DSLs.

2.1 Definition of Domain Specific Languages

To establish boundaries to a term with a generally vague meaning, Martin Fowler defines DSLs as follows [1]:

Domain specific language: a computer programming language of limited expressiveness focused on a particular domain.

A DSL is therefore characterized by first being *a computer programming language*. Its primary usage is to allow humans to instruct a computer to perform a certain action. Contrary to a *general-purpose language* like Java or Ruby, however, a DSL only has a *limited expressiveness* and is specialized on a *particular domain*. In other words, a DSL only supports a small amount of features and syntax which are tailored to the domain where it should be employed.

Fowler furthermore distinguishes DSLs into three categories [1]:

- **External DSLs** are separate from the main language of the application and usually have a custom syntax. They therefore have to be parsed by the host application in order to execute them.
- **Internal DSLs** use capabilities of the general-purpose language of the application to try to offer the feeling of a custom language. The code of the DSL is valid code in its general-purpose language as well, so no additional parsing is necessary.
- **Language Workbenches** offer environments for defining and building DSLs as well as writing scripts for the DSLs. Since language workbenches do not play any role in this thesis, they will not be given further attention.

For all these types of DSLs, the boundary which determines whether something is or is not a DSL is quite blurry. For internal DSLs, the distinction has to be made between a normal *application programming interface* (API) and an actual internal DSL. For Fowler [1] the difference lies in the nature of a DSL to define a new language in form of a grammar. The documentation of an API can offer a good indication whether the module or library exposes a normal API or a DSL. In the case of APIs, methods usually can be documented by themselves and therefore have a self-sufficient meaning. In a DSL, however, methods usually do not hold any meaning by themselves but can only be interpreted in context of a larger expression.

Listing 1 depicts a testing library offered by the Spring framework to check the behaviour of a RESTful backend. The testing library offers a variety of static methods combined with elegant method chaining to fluently define a test. For instance, the *andDo* method expects an

```
apiTestClient.perform(get("/users"))
  .andDo(print())
  .andExpect(status().isOk());
```

Listing 1: The Spring framework offers internal DSLs for testing purposes.

object that implements a *ResultHandler* interface. The static method *print* constructs such an instance and passes it to the *andDo* method. It is therefore apparent that a standalone executing of the *print* method would not result in anything meaningful. The *print* method as well as the *andDo* method can only be reasonably evaluated when they are both combined with each other.

```
public class PersonBuilder {
    private String name;
    private Integer age;
    private String placeOfBirth;

    public static PersonBuilder newPerson() {
        return new PersonBuilder();
    }

    public PersonBuilder name(String name) {
        this.name = name;
        return this;
    }

    public PersonBuilder age(Integer age) {
        this.age = age;
        return this;
    }

    public PersonBuilder placeOfBirth(String placeOfBirth) {
        this.placeOfBirth = placeOfBirth;
        return this;
    }

    public Person build() {
        return new Person(this.name, this.age, this.placeOfBirth);
    }
}

// Usage
PersonBuilder.newPerson()
    .name("John Doe")
    .age(21)
    .build();
```

Listing 2: It is arguable whether *PersonBuilder* can be considered to be a DSL.

On the other hand, listing 2 shows the definition and usage of a *builder pattern* to create instances of a hypothetical *Person* class. In this case it is arguable whether *PersonBuilder*

exposes an internal DSL. Each method of the builder such as *name* or *age* can be independently described by setting an attribute of the resulting person i.e., each method has a self-sufficient meaning by itself. Additionally, except having to call *newPerson* at the beginning and *build* at the end, the creation of a new person is not dependent on any grammar which an actual language should be composed of.

For external DSLs one has to differentiate between a DSL and a general-purpose language, though the boundary is not as blurry as with internal DSLs. A good example presented by Martin Fowler [1] is the *R language*[1], which is a programming language for statistical computing. While focusing on a particular domain, the R language is not limited in its expressiveness and can be employed for purposes it was not initially intended for. Therefore, though it partly complies with the definition of a DSL, it should be categorized as a general-purpose language. A popular and widely spread example for an external DSL is *regular expressions*. It is specialized on matching text and offers only the amount of features and syntax to excel for this purpose. As a general rule, external DSLs are not *Turing-complete*. They usually do not offer mechanisms for control flow such as loops or conditions combined with the possibility to define variables and functions.

2.2 Benefits and Problems of Domain Specific Languages

After defining and categorizing DSLs, the question arises why developers of software systems should actually build and use DSLs. What are potential benefits as well as problems of DSLs? By weighing each of the advantages and downsides of DSLs, software professionals will be able to decide whether a DSL could potentially help to solve a certain problem or not.

According to Martin Fowler, DSLs offer the following advantages [1]:

- **Improving Development Productivity:** Since DSLs are specialized to express a certain aspect of a system, the code of the DSL will be more easy to write, read, and understand. This leads to an improvement of productivity by both making less mistakes as well as fixing defects more quickly. In Fowler's words: "The limited expressiveness of DSLs makes it harder to say wrong things and easier to see when you've made an error.".
- **Communication with Domain Experts:** Good communication in software projects is, according to research, a very important critical success factor for projects to succeed [2]. Since software professionals develop systems for a wide variety of industries, they have often to be in contact with experts of the particular industry, so called *domain experts*. Due to their specialized syntax, DSLs offer the possibility for domain experts to read and correct source code and therefore highly improve the communication between tech and non-tech project stakeholders.
- **Change in Execution Context:** A common reason for external configuration files written in XML and similar formats, is the ability to read and evaluate them at runtime. This way the system does not have to be recompiled for every change of its configuration. DSLs offer a resembling advantage: they can also shift changes of logic of a system from compile time to the execution of that system.

[1] `https://www.r-project.org/`

- **Alternative Computational Model:** Most general-purpose languages follow the *imperative style* of computation: the computer is told what to do in a certain sequence with features such as control flow and variables. For some problems, however, different approaches are more suitable and easier to utilize. Build automation is one of these problems: build tools such as *Apache Maven*[2] generally offer a *declarative style* to describe the build of a software system. Instead of focusing on *how* something should be done, the declarative style of programming concentrates on *what* should happen, leaving the *how* to a different layer of the system. When creating and using DSLs, it is also possible to employ a different computational model than the main language of the application with which it is easier to express or define certain aspects of the respective domain.

Contrary to these advantages, the usage of DSLs also comes with some problems and threats. Among them are the following [1]:

- **Language Cacophony:** This term was coined by Martin Fowler and states that learning new languages is generally hard. Therefore, it is apparent that combining multiple language for a project complicates the development compared to only using a single language. It is therefore necessary to determine whether or not learning a DSL is less costly opposed to understanding and working at the problems at hand without a DSL.

- **Cost of Building:** The most obvious problem of creating a DSL is the initial cost of building it. However, not only the initial costs of implementing the DSL has to be taken into account. Throughout time the DSL will have to be maintained and extended as well. Moreover, according to Fowler, it is not common for developers to know the techniques which are necessary to build DSLs which further aggravates the cost of implementing one.

- **Ghetto Language:** With the *ghetto language problem*, Martin Fowler refers to an issue which contrasts with the language cacophony problem. The term describes a language, built in-house, which is being utilized in more and more systems of the company as well as being continually extended with features and therefore slowly evolving into a general-purpose language. In the long run, this will lead the company to be inflexible regarding technological innovations and shifts in the industry as well as making it harder to hire staff. As a consequence, companies should clearly define the purpose and boundaries of their DSL and refrain from breaching these decisions.

- **Blinkered Abstraction:** The last problem Fowler highlights, is the situation where developers are too confident about their DSL and try to fit the world to work with their language, instead of changing the language in accordance to the world. Thus, software professionals must view their DSL to be constantly under development, instead of regarding it as being finished.

As a conclusion, there are two possible reasons not to use a DSL. First, in case none of the benefits of a DSL applies to the problem at hand it is naturally not a fitting tool to solve that problem. Secondly, if the costs and risks of building a DSL outweigh its potential benefits. Otherwise it can be worthwhile to consider building or using a DSL to benefit from the potential prospects as set out in this section.

[2] `https://maven.apache.org/`

2.3 Implementation of Domain Specific Languages

3 Overview of GraalVM

3.1 Motivation

Why do we need Graal?
Write more of Java in Java itself.

3.2 Features

3.2.1 GraalVM Compiler

Explanation and some benchmarks
source: `https://www.youtube.com/watch?v=sFf15TvSXZ0`

1. What is a JIT compiler

   When compile Java Source with javac –> Java Bytecode At Runtime –> Bytecode is compiled in Machine Code Machine Code delivers usually much better perf

2. Why write JIT compiler in Java

   C2 is the JIT compiler written in C++ Developers of JIT think C2 is too old now, too hard to maintain Developing in Java tends to be easier and more productive than in C++

3. JVM compiler interface

   Allows to plugin a custom JIT compiler for the JVM written in Java In thesis interface can shown here: `https://github.com/openjdk/jdk/blob/master/src/jdk.internal.vm.ci/share/classes/jdk.vm.ci.runtime/src/jdk/vm/ci/runtime/JVMCICompiler.java` Takes bytecode and returns new bytecode

4. Graal JIT compiler process

   The compiler first represents code in graphs Every node will then be transformed into machine code

5. Optimisations

   - Basically changes the nodes
   - Canonicalisation: e.g. –x –> x
   - Global value numbering: remove redundant code: (a + b) * (a + b) -> only (a + b) once calculated
   - Lock coarsening: two synchronized locks immediately after each other -> change to only once

3.2.2 Native Images

Explanation and some benchmarks yet again

3.2.3 Truffle Framework

Basic explanations: why is there a truffle framework and what is achievable

3.2.4 Polyglot Applications

Basic explanations: what's possible here

4 Domain Specific Languages in GraalVM

4.1 Technical Overview

How to build DSLs with GraalVM?

4.2 <INSERT NAME OF DSL>

Introduce the DSL of this thesis here 1 2 3 4 5

$$A = \{ x \mid x \in (A \cap B) \} \tag{1}$$

4.3 Implementation of <INSERT NAME OF DSL>

Highlight some key aspects of implementation

4.4 Evaluation

Evaluate the DSL and GraalVM, highlight pain points etc.

5 Integration of Domain Specific Languages

5.1 Technical Overview

How do polyglot applications technically work?

5.2 Integration of <INSERT NAME OF DSL>

Showcase how it's done using the thesis DSL

5.3 Evaluation

Evaluation how good this actually works

6 Conclusion

Business as usual

References

[1] Martin Fowler. *Domain-Specific Languages*. Pearson Education, 2010.

[2] Goparaju Purna Sudhakar. A model of critical success factors for software projects. *Journal of Enterprise Information Management*, 2012.