An Evaluation of Scala as a host language for DSLs

Reinhard Pointner,
BSc (Hons) Computing

Supervisor: Bogdan Matuszewski
Second Reader: Chris Casey
27 June 10

Abstract

This paper discusses and evaluates the use of the Scala programming language in regard to domain-specific language development. This paper will outline the various advantages DSLs provide in general and explain the difficulties inherent in DSL development. In addition we will examine different approaches to DSL development and their respective characteristics. We will then evaluate the features of Scala DSL development. These features include easy reuse of existing language infrastructure, composability of multiple DSLs and pluggable DSL implementation due to the fact that DSLs are defined within the Scala language itself as well as performance due to compilation. Support for domain-specific languages is then further discussed by looking at Groovy in contrast to Scala including a brief explanation of how Scala can be used as a foundation for an external DSL. We will then look at how Scala can be used for the creation of a domain-specific language for human animation and some of the benefits of this approach.

1 Introduction

1.1 Domain-Specific Languages

Mernik (2005) defines DSLs as languages that are designed and optimized for a specific purpose or set of tasks, thereby offering substantial gains in expressiveness and ease of use in comparison to general-purpose programming languages. The idea of domain-specific languages is not new. DSLs for various tasks have been around since early ages of computing (Mernik, 2005). One of the main differences between GPLs and DSLs is that the latter is not necessarily executable by itself. It may merely be some kind of description language like regular expressions for text patterns, make files for building software or simply HTML and CSS for websites. Although there are many successful and widely used examples of DSLs, standalone applications rarely make use of small special-purpose languages. Mernik (2005) and Hofer, et al. (2008) agree that the primary reason for the small adoption of DSLs is simply that the extreme amount of effort that is required to create a new language from scratch almost always outweighs possible benefits.

1.2 Overview

This paper will introduce DSLs in general and evaluate Scala as a host language in particular. The following section will outline on various approaches to DSL development. The third section will evaluate the use of Scala in respect to certain set of desirable DSL development features and move on to a direct comparison between Scala and Groovy. The use of Scala for writing a DSL for human animation will be discussed in the fourth section.

2 Approaches to DSL development

Fowler (2009) generally distinguishes between internal and external DSLs. External DSLs are usually written from scratch as part of an application that uses this DSL for interpreting certain types of input. An internal DSL on the other hand is built upon or within an existing language. The aim is to reuse existing language infrastructure like syntax checker, parser, compiler and interpreter as far as possible.
Mernik (2005) points out however, that DSL development does not only require technical expertise in language design but also domain expertise and that few people have both. Since DSLs are typically designed to be used by non-programmers to accomplish a specific range of tasks, a deep understanding and insight into that purpose is required when defining syntax and semantics.

2.1 External DSLs

Mernik (2005) talks about language invention in this regard as external DSLs are typically implemented from scratch. Language design can be arbitrary and there are virtually no limitations which allows for syntax that is close to the notations used by domain experts (Mernik, 2005). This requires defining a grammar, then a parser for converting the textual or graphical input to some kind of data structure and finally an interpreter to evaluate the input expression. Therefore the implementation will require a considerable amount of time, effort and technical expertise in language design. There are many tools like parser generators that help substantially, but language development will nevertheless remain quite complex. It is for that reason that extensibility of the language is usually not the first priority. Mernik (2005) highlights this as one of the main disadvantages of this approach and argues that domain-specific languages are generally much more prone to language changes than general purpose languages because the domain experts will usually request more and more features as they are working with the language. In addition maintaining the language will proof to be difficult especially for developers who are not familiar with the specific implementation.

2.2 Internal DSLs

Internal or embedded DSLs are typically built within an existing language. The DSL will be able to reuse the complete tool chain of the host language. Therefore no grammar, parsers or any other language development tools are required. The language is defined solely by using language features of the host language which imposes certain limitations to the DSL syntax. Mernik (2005) marks this as one of the main disadvantages and argues that most languages don’t allow for arbitrary syntax extension. Naturally languages that provide syntactic flexibility are more suitable for creating a DSL. One of the main advantages of internal DSL development is that it does not require any kind of special expertise in language development. Fowler (2009) suggests the term “fluent interface” in this regard, because internal DSLs are fundamentally a form of API. This is also one of the reasons why internal DSL are much easier to maintain and to extend because in essence developers only need to be familiar with the host language to be able to improve upon the DSL.
Mernik (2005) however defines this type of DSL in more general terms when he talks about language exploitation. When reusing an existing language the DSL can overcome limitations of the host language by either extending the host language itself or by using some form of pre-processor that converts DSL code to host language code before compilation. This approach of adapting an existing language to make it more expressive will make language development significantly more complicated when compared to the pure embedding approach that solely uses existing language features.

3 Evaluation of Scala as a host language

In the previous section the merits of domain-specific languages have been explained and general implementation approaches have been discussed. The idea of using Scala as a host language puts this approach into the domain of internal DSL development and therefor most of the previously mentioned advantages and disadvantages apply.
This paper focuses on Scala rather than any other language because Scala promises language flexibility that is intended to allow developers to capture their respective problem domains more naturally (Odersky, et al. 2006). The same kind of reasoning leads to domain-specific languages.
Mernik (2005) argues that embedding suffers from limited user-definable syntax in languages like Java and observes no trend towards more powerful languages. However Hofer, et al. (2008) illustrates the opposite with Scala compellingly but points out that there are certain syntactic constructs that cannot be modelled easily.

Hofer, et al. (2008) identifies the following set important DSL features:

- Reuse of Infrastructure
- Pluggable semantics
- Performance
- Composability

3.1.1 Reuse of Infrastructure

The approach of defining a DSL purely in Scala syntax immediately allows the language developer to reuse the complete Scala API as a well as all the additional tools. This includes syntax checker, compiler and maybe most notably IDE support. There is also debugging support but since the debugger will work on the host language level rather than the DSL level it is less useful (Hofer, et al. 2008). It is noteworthy however that debugging support is generally not implemented at all in most DSLs because it is usually not considered worth the effort.

3.1.2 Pluggable semantics

The idea of pluggable semantics is to decouple the DSL syntax from a single specific interpretation and thereby allowing multiple possibly very different implementations of the same language. Scala is perfectly suited for this purpose as this concept can be implemented using straightforward polymorphism as Hofer, et al. (2008) illustrates. Essentially the DSL syntax is defined by an abstract class definition and the concrete semantics are defined by the respective implementations thereof. Those implementations themselves may then again be extended and adapted by others. Where Mernik (2005) was not able to answer how to extend implementations in a safe and modular manner, Hofer, et al. (2008) seems to have found a simple solution with this approach.

3.1.3 Performance

Performance is generally not an issue for domain-specific languages. However it might be noteworthy that any Scala DSL code is compiled to Java byte code, virtually eliminating language related overhead.

3.1.4 Composability

Composability describes the concept of using multiple DSLs in the same code. This is virtually impossible for external DSL implementations because it would require merging two or more very different and possibly extensive codebases.
As far as Scala is concerned, DSLs are a merely a form of API. When implementing a new DSL the developer can simply make use of existing lower-level DSLs. Multiple DSLs may even work and interact on the same level as long as they agree on common interfaces and types.

3.2 Scala vs Groovy

Scala is only one of many new languages targeting the JVM. Each one of them tries to solve issues that are inherent in traditional programming languages. Scala and Groovy are among the most prominent ones.
Scala is a statically-typed compiled language. Groovy on the other hand is a dynamically-typed scripting language. Because of those features, it seems apparent that Groovy would be a far better choice as a host language for both internal and external DSLs. It is easy to evaluate Groovy expressions dynamically at runtime from within a Java application and languages targeted at non-programmers usually don't make use of type systems because it would unnecessarily increase complexity. In fact, G. Laforge (2007), one of the core developers of Groovy, argues that Groovy syntax is malleable and flexible which makes it the perfect choice for creating DSLs. However Groovy first and foremost improves upon Java and adds many features of popular scripting languages. The aim of Scala on the other hand is to create a new language that is in itself extensible enough in order to allow developer to model their respective domains easily and naturally in libraries and frameworks (Odersky, et al. 2006), instead of providing a certain set of built-in features. Even though Scala is statically typed it does not hinder DSL development significantly because of a sophisticated type inference system which permits to omit actual type (Odersky, et al. 2006). This means that the type does not have to be stated explicitly in most cases giving Scala many syntactic advantages that are usually associated with dynamically typed languages. Since Scala is a compiled language it stands to reason to question if it sensible at all to use Scala as a foundation for an external DSL because user-defined code will need to be evaluated dynamically at runtime. But this is actually only a minor issue because Scala provides API for dynamic compilation and interpretation. However Scala is more complicated than Groovy in this regard, because it is not primarily designed to be used for that purpose.
Even though Groovy seems more suitable to be used as a foundation for an external DSL initially, Scala extensive support for growing the language and internal DSLs, which can be exposed as external DSLs, may make it more worthwhile in the long run.

4 DSL for human animation

One aim of the project is to design and implement a simple user-friendly domain-specific language for scripting a virtual 3D avatar. Z. Huang, et al. (2003) tackled a similar problem when implementing STEP (Scripting Language for Embodied Agents) and chose to expose the internal logic via a DSL to the user. However due to the simplicity of the language some use-cases that would have been possible internally are not accessible via the external DSL.
A Scala DSL could help solve the dilemma of trying to simplify the input language and make it user-friendly while allowing advanced and much more complicated code as well. Since anything will be valid Scala code the user can choose to solely use the simplified DSL but may also directly access any language constructs, classes or APIs that are available to the programmer. Interestingly this concept is applicable vice versa. The developer may also choose to use the DSL in internal code. Therefore more complex parts of the DSL implementation could be defined in basic constructs of the very same DSL. The language definition can also be considered as an independent software component and may be reused for scripting any human-shaped body. However one of most significant reasons for using Scala in any kind of project is simply that a lot can be achieved within a limited amount of time.

5 Conclusion

The Scala language is DSL friendly by design. One of the major goals of Scala is to allow developers to produce expressive and concise code as well as easy-to-use APIs in form of internal DSLs. It is possible to implement DSLs in a modular manner that is impossible for external DSLs. First and foremost all DSL code is valid Scala code. Therefore multiple DSLs can be used in conjunction or on top of each other. In addition the DSL definition itself is a form of class that may be extended allowing multiple implementations of the same DSL using simple polymorphism.

As mentioned before, Scala is a compiled language and as such it is not primarily designed to be used as the foundation of an external DSL. There are however approaches of using the Scala compiler API to interpret new input dynamically at runtime.

Since everything is hosted completely within Scala, there is no actually language development in the traditional sense involved when implementing the DSL, allowing for rapid DSL development as well as straightforward maintenance and language extension.

6 References

C. Hofer, K. Ostermann, T. Rendel, A. Moors, 2008. Polymorphic Embedding of DSLs.
In: GPCE (Generative Programming and Component Engineering),
7th international conference on Generative programming and component engineering.
Nashville, Tennesse, USA, 19-23 October 2008. ACM.

M. Mernik (2005). When and How to Develop Domain-Specific Languages.
ACM Computing Surveys. 37 (4), 316-344.

Martin Fowler, 2009. Domain Specific Languages (WORK-IN-PROGRESS) [online]
(Updated 30 Jun 2009) Available at: http://martinfowler.com/dslwip/index.html
[Accessed 10 November 2009]

M. Odersky, P. Altherr, V. Cremet, I. Dragos, G. Dubochet, B. Emir, S. McDirmid, S. Micheloud, N. Mihaylov, M. Schinz, L. Spoon, E. Stenman, and M. Zenger (2006).
An Overview of the Scala Programming Language. 2nd ed. Lausanne: EFPL (École Polytechnique Fédérale de Lausanne). p18.

G. Laforge and J. Wilson, 2007. Tutorial: Domain-Specific Languages in Groovy [online]
(Updated 7 March 2007) Available at: http://glaforge.free.fr/groovy/QCon-Tutorial-Groovy-DSL-2-colour.pdf
[Accessed 15 November 2009]

Z. Huang, A. Eliëns, C. Visser. Implementation of a Scripting Language for VRML/X3D-based Embodied Agents. In: Web3D
8th international conference on 3D Web technology
Saint Malo, France, 09-12 March 2003. ACM.