UNIVERSITY OF OSLO
Department of Informatics

Embedding Efficient DSLs on the JVM

A review of alternative languages

Master thesis

Eivind Barstad Waaler (*eivindwa*)

May 2, 2010

Abstract

The Java Virtual Machine (JVM) is a popular software platform capable of running programs in the intermediate language called Java bytecode. The JVM supports a wide range of programming languages with different approach to typing, programming paradigm and compilation. This thesis examines some of these languages, with a focus on their support for embedding domain-specific languages. It is demonstrated how various languages provide different possibilities, with regard to syntax and semantics, customization, usage and performance. Examples are given showing what programming languages are suited to particular domain-specific tasks, and further research is suggested in the form of new language features and new ways to utilize existing language features.

Contents

Abstract 1

Contents 3

List of Tables 5

List of Figures 5

List of Code Listings 6

Preface 7

1 Introduction 9
- 1.1 Problem Description 9
- 1.2 Method 10
- 1.3 Domain-Specific Languages 11
- 1.4 Summary of Requirements and Report Structure 13

2 Programming Languages and Examples 15
- 2.1 JVM Programming Languages 15
- 2.2 Examples 21
- 2.3 Other Examples Used 25

3 Syntax and Semantics 29
- 3.1 Function Chaining/Nesting 29
- 3.2 Static Methods/Singleton Objects 30
- 3.3 Member Imports 31
- 3.4 Method Names and Operator Notation 32
- 3.5 Special Syntax Constructs 33
- 3.6 Type Inference 35
- 3.7 Implicit Type Conversion 36
- 3.8 By-name Parameters 37
- 3.9 Lambda Expressions/Anonymous Functions 37
- 3.10 Anonymous Types 39
- 3.11 Object Initializer 39
- 3.12 Metaprogramming 40
- 3.13 Open Classes/Monkey Patching 43
- 3.14 Dynamic Dispatch – Method Missing 43
- 3.15 Summary 44

4 Creating Customizable DSLs 45
- 4.1 Specializing a General DSL 45
- 4.2 Combining DSLs 49
- 4.3 Summary 52

5 Changing the Host Language 55

5.1 Removing from the Host Language – Pruning . . . 55
5.2 Extending the Host Language . . . 58
5.3 Summary . . . 60

6 Exploring Ways to Use Embedded DSLs **61**
6.1 API/Library . . . 61
6.2 Scripting . . . 62
6.3 Interactive Console – Interpreter . . . 63
6.4 Embedded Interpreter . . . 64
6.5 Tool Support . . . 64
6.6 Error Handling . . . 65
6.7 Summary . . . 68

7 Performance **69**
7.1 Host Language – Dynamic vs. Static Typing . . . 69
7.2 Code Efficiency . . . 70
7.3 Concurrency – Parallel or Distributed Computing . . . 73
7.4 Secondary Processors – GPU . . . 78
7.5 JVM Tuning – JVM Arguments . . . 78
7.6 Summary . . . 79

8 Summary **81**
8.1 Conclusion . . . 81
8.2 Suggestions for Future Work . . . 84

Bibliography **87**

A Downloading and Building the Examples **89**
A.1 Viewing the Code Online . . . 89
A.2 Downloading and Running the Examples . . . 89

List of Tables

2.1 Programming languages comparison matrix. 16

8.1 Overview of tasks and suited programming languages. 84

List of Figures

1.1 Illustration of method phases. 11

2.1 Visual representation of the relationship between Scala and Java. 18
2.2 Example comparing syntax consistency of Clojure and Java side by side. 21
2.3 Example image before and after blurring with average filter. 22
2.4 Visualising the process of blurring an image using average filtering with a 3x3 neighbourhood. . . . 23
2.5 Example pie chart created with the JFreeChart based charting DSL. 26

4.1 Two different images with generated histograms 53

6.1 Screenshot from Kojo, a programming learning environment with an embedded DSL-REPL. 65
6.2 Screenshot from IntelliJ IDEA, an IDE having good support for Scala. 66

7.1 Mini benchmark – implementation in 5 languages. 70

List of Code Listings

2.1 Java jMock example. 17
2.2 Scala introductory example. 18
2.3 Groovy “hello world” example. 18
2.4 Groovy anonymous function example. . . 19
2.5 Ruby “hello world” example. 19
2.6 Ruby Java integration example. 20
2.7 Clojure Fibonacci example. 20
2.8 Clojure – Java integration. 21
2.9 MATLAB® image processing example. . . . 22
2.10 Scala image processing example. 22
2.11 Scala image processing syntax example. . . 23
2.12 Scala companion object example. 23
2.13 Scala image processing traits. 24
2.14 Scala statistics DSL 25
2.15 Scala statistics DSL usage. 25
2.16 Scala charting DSL 25
2.17 Ruby on Rails `ActiveRecord` example . . 27
2.18 Java Hibernate example 27
3.1 Java function chaining example. 29
3.2 Java method chaining and nesting. 30
3.3 Scala traits providing DSL order. 30
3.4 Java static sort example. 31
3.5 Scala singleton object sort example. 31
3.6 Scala image processing singleton objects. . 31
3.7 Java static member import example. 31
3.8 Scala member import example. 32
3.9 Scala image processing member imports. . 32
3.10 Scala operator notation function example. 32
3.11 Scala ScalaTest example. 33
3.12 Ruby parameter example. 33
3.13 Scala `apply` method example. 34
3.14 Scala `apply` method factory example. . . . 34
3.15 C# LINQ example. 35
3.16 Scala `for comprehension` example. . . . 35
3.17 Clojure relational algebra example. 35
3.18 Scala basic type inference example. 35
3.19 Ruby/Groovy variable definitions. 36
3.20 Scala type inference example. 36
3.21 Scala implicit type conversion example. . . 36
3.22 ScalaTest implicit conversion example . . 37
3.23 Scala by-name parameter examples 37
3.24 Java simple for-loop/if example 38
3.25 Scala basic anonymous function example 38
3.27 Scala IP anonymous functions 38
3.26 Scala anonymous function example. 39
3.28 Scala anonymous types example. 39
3.29 C# LINQ example using anonymous types. 39
3.30 Clojure data structure example. 39
3.31 C# LINQ object initializer example. 40
3.32 Scala object initializer example 40
3.33 Java metaprogramming with annotations. 41
3.34 Groovy metaprogramming example. 41
3.35 Ruby on Rails dynamic metaprogramming 42
3.36 Ruby dyn. metaprogramming example . . 42
3.37 Simple Clojure macro example. 43
3.38 Ruby “monkey patching” example 43
3.39 Ruby `should` method added to String. . . . 43
3.40 Ruby on Rails `ActiveRecord` example. . . 44
4.1 Scala traits example. 46
4.2 Groovy mixing composition example. . . . 46
4.3 Ruby mixin composition using modules. . 47
4.4 Matrix example using type parameters. . . 48
4.5 Image example using virtual type. 48
4.6 Clojure multi-method example 49
4.7 Scala statistics DSL basic example. 49
4.8 Scala word statistics example. 50
4.9 Scala implicit conv. DSL combine example 50
4.10 Scala image histogram example 51
4.11 Package template DSL example. 51
5.1 Scala “Divide-by-zero” compiler plugin. . . 56
5.2 Ruby String pruning example. 57
5.3 Ruby `undef_method` example. 57
5.4 Clojure redefining core functions. 57
5.5 Clojure undefining core functions. 57
5.6 Clojure macro binding core variable. . . . 58
5.7 Scala built-in keyword functions 59
5.8 Clojure macro adding unless 59
5.9 Clojure macro reverting statements order. 60
5.10 Clojure reverse statements example 60
6.1 Java example using jMock as an API. 61
6.2 Scala example using jMock as an API. . . . 62
6.3 Scala scripting usage example. 62
6.4 PreDef file with IP imports 63
6.5 Scala REPL execution 63
6.6 Scala REPL image processing example. . . 63
6.7 Scala exception example. 67
6.8 JRuby exception example. 67
6.9 Clojure exception example. 67
6.10 Scala control-abstraction example. 67
6.11 Clojure with-open example. 68
7.1 Scala matrix list of lists 71
7.2 Scala matrix 2D array 71
7.3 Scala matrix neighbour average 72
7.4 Scala matrix `for-comprehension` 73
7.5 Scala matrix `while-loops` 74
7.6 Scala actor executor 75
7.7 Scala futures example 76
7.9 Scala parallel image processing operations. 76
7.8 Scala futures parallel mechanism 77
7.10 Clojure memoization example 78
7.11 Add two arrays of integers using ScalaCL. . 79

Preface

This is a master-thesis of 60 credits in the field of Informatics. It was supervised at the Object-orientation, Modelling, and Languages research group[1] at the Department of Informatics, Faculty of Mathematics and Natural Sciences, University of Oslo.

I want to thank my wonderful wife Anniken for the support and understanding she has shown during this process.

My thanks and thoughts also go to the memory of my grandfather Bjarne A. Waaler. His constant search for knowledge and truth has been an enormous inspiration all my life. I am proud to make a contribution to the same university where he spent so many of his years working, both in research and administration.

At last, but not least, I would like to thank my supervisor Professor Birger Møller-Pedersen at the Department of Informatics, UiO. His unique experience from projects such as the BETA programming language and the SWAT project has proven extremely valuable for me. Through many good discussions at his office I've learned much about current and historic programming language concepts. His enthusiasm has motivated me from beginning to end.

EIVIND BARSTAD WAALER
Oslo, Norway
May 2010

[1] http://www.ifi.uio.no/research/groups/oms/

Chapter 1

Introduction

The goal of this chapter is to describe the problem as detailed as possible. It has taken me a long time to clarify and understand the problem, and as such I think it is justified to spend some time and space describing the background and why it is important. First the problem formulation is given, with a brief background and motivational description. The rest of the chapter describes the method followed, and gives more detailed explanations of the terminology and background of choices made regarding the problem formulation. The final section gives a brief outline of how the rest of the report is structured.

1.1 Problem Description

1.1.1 Problem Formulation

Evaluate some major alternative programming languages on the Java Virtual Machine (JVM) with regards to embedding domain-specific languages. Identify important requirements when working with embedded domain-specific languages and show what programming languages are best suited.

1.1.2 Background

A domain-specific language (DSL)[2] is a programming (or modeling) language dedicated to a particular problem domain. This thesis looks at embedding programming DSLs into existing general-purpose programming languages, with a focus on several categories of languages with different features, and how these features are used. The scope is limited to the platform offered by the JVM, and a set of programming languages (described further in chapter 2 on page 15). A more detailed definition of domain-specific languages, with a discussion around scoping and requirements, is given in section 1.3 on page 11.

1.1.3 Motivation

I have 10 years experience using the Java programming language, working as a professional consultant for BEKK[3]. I have always been fascinated by various programming languages, and how they can be used to approach problems and solutions. When starting on a master thesis it is natural for me to focus on programming language capabilities. I feel that the embedded DSL domain provides a good basis for exploring different categories of programming languages, and by limiting the focus to the JVM I can control the scope and draw on my experience working with the Java platform.

[2] Also known as problem-oriented or special-purpose programming language in computer literature.

[3] Bekk Consulting AS (`http://www.bekk.no`).

1.2 Method

This section describes the research method applied in the thesis. In [7] the concept "Evidence-Based Software Engineering" (EBSE) is introduced. It describes a method that can be used when making decisions in Software Engineering, combining evidence found in literature with practical experience and conducted experiments. Although the method is mainly targeted towards practitioners looking to support decision-making it can easily be adapted to support an evaluative thesis as this one. The main steps of EBSE are as follows:

1. Convert a relevant problem or need for information into an answerable question.
2. Search the literature for the best available evidence to answer the question.
3. Critically appraise the evidence for its validity, impact, and applicability.
4. Integrate the appraised evidence with practical experience and the client's values and circumstances to make decisions about practice.
5. Evaluate performance in comparison with previous performance and seek ways to improve it.

Inspired by the EBSE method, I have created my own method suited to solve the problem presented in the previous section. It is a set of steps that can be run in iterations, each bringing further results, that is suited for a time-boxed assignment (as a master thesis). The steps followed are illustrated in figure 1.1 on the facing page, and described in the list below:

1. **Identify requirement** – Identify requirements that are relevant in solving the problem. Break down big requirements into smaller ones to reach a "solvable" size.
2. **Search theory** – After choosing a requirement a theory search is conducted to find relevant research and information. From the theory found decide if separate experiments are needed, or if a preliminary conclusion can be made directly.
3. **Implement example** – Implement example code to prove or show how the requirement can be solved.
4. **Measure** – Some of the requirements are directly measurable (for example "which programming language is fastest in adding two integers?"). This step makes sure that relevant results are measured where possible.
5. **Conclude** – Make a preliminary conclusion based on the results found so far. Identify all results found, and possibly further requirements to find out more.
6. **Evaluate** – Evaluate results and method continuously. Before starting work on a new requirement improve those parts of the process not being efficient or useful.

The whole process starts by identifying some overall requirements related to the problem formulation. These requirements are then broken down into smaller ones. New requirements are discovered in the process and can be prioritized when starting new iterations of the process.

A Note on Theory/Literature

Many of the programming languages used in this thesis are relatively new, and not commonly studied in much academic research yet. Much of the most recent material written on or about the languages exist in less formal ways, typically as Internet articles or blog posts. I have tried to use scientific research papers as theory basis as much as possible, but a certain amount of references/footnotes will also refer to links on the Internet. Although not as reliable as scientific resources, I find it more honest to include the source of any ideas/inspiration. As a general rule I have tried to put URL sources directly as footnotes, while more scientific resources are used as references.

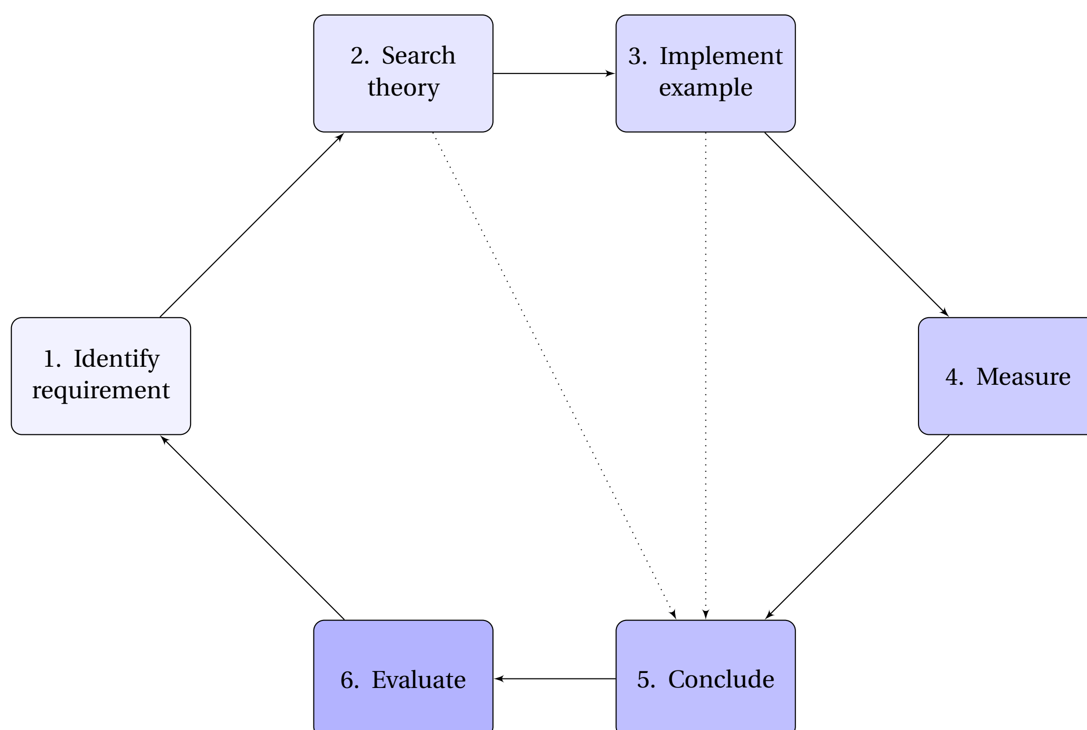


Figure 1.1: Illustration of method phases.

1.3 Domain-Specific Languages

"Domain-specific languages (DSLs) are languages tailored to a specific application domain. They offer substantial gains in expressiveness and ease of use compared with general-purpose programming languages." [27]

DSLs are often categorized in two major types; external or embedded (also referred to as internal) DSLs. An external DSL is a separate programming language with custom syntax and a full parser. Some well-known examples of external DSLs are SQL (domain of database queries), CSS (domain of HTML styles) and XML (domain of data exchange). An embedded DSL, on the other hand, is created in a general purpose programming language, often referred to as a "host" language. Various capabilities in the host language are exploited to create interfaces that look and behave almost like a particular programming language. So the syntax and semantics of the DSL are defined only using the syntax and semantics already present in the host language. Atkey et al. [1] describe two principal advantages with embedded over external DSLs. First, the work needed by the programmer to implement the DSL is reduced (as a separate parser and interpreter does not have to be implemented), and it is easier for the user who does not necessarily need to learn an entirely new language. Second, in integration with the host language and other DSLs implemented in the same language (as integration of two DSLs written in the same language is essentially the same as using two different libraries in the same language together). The boundary between external and embedded DSLs is not necessarily a fixed one. Elements from both categories can be mixed in a single DSL implementation, as shown by Bravenboer and Visser in [3].

In this thesis I have focused solely on embedded DSLs running on the JVM. As described in the book «DSLs in Action» by Debasish Ghosh [16], several different programming languages can be used to implement embedded DSLs on the JVM. In this thesis I have identified some different requirements of embedded DSLs, and then looked at various JVM programming languages (described in section 2.1 on page 15) in order to find out what languages are best suited for the different requirements. The rest of this chapter describes these requirements, with a discussion of why I think they are important.

1.3.1 Embedded DSL or just Advanced Library Design?

It is difficult to draw a precise line between an embedded DSL and a regular library. If you have a library specialized to solve problems in a given domain, is that enough to call it an embedded DSL? How much "special syntax" do you need to include in the library to make it a DSL? What kind of syntax qualifies for DSL use? Answering questions like these is difficult, and in my opinion not contributing to a fruitful discussion. I have tried to turn my focus toward interesting capabilities different languages have, and the problems they solve.

Martin Fowler and Eric Evans have defined the name "fluent interface" [11] to describe interfaces which provide a more readable and "flowing" syntax. Many of the examples given of fluent interfaces are small libraries solving tasks in a specific domain, and as such are also good examples of embedded DSLs. Eric Evans has also written a book [9] and held a number of talks on the subject of Domain-Driven Design (DDD), a way of thinking and prioritizing when working with software projects that deal with complicated domains. DDD hardly covers the subject of embedding DSLs, but I feel that a lot of the DSL design is focused on understanding and expressing a particular domain and as such has a lot in common with the thoughts behind DDD.

I believe the main requirement of an embedded DSL is the way it looks and feels for a domain expert using it. This leads to the first requirement in the list, looking at what syntax and semantics the different programming languages support making them suited in building the syntax and semantics required by a particular embedded DSL. Much of the existing DSL research is concerned with this requirement, looking primarily at the syntax and semantics of host languages. There are, however, other requirements that may be important for particular DSLs, as described in the following sections.

1.3.2 Creating Customizable DSLs

There are situations where having the ability to customize a DSL may be useful. There are also many different ways that the DSL can be customized or in other ways changed to suit a specific need.

A way to provide customization is to implement the DSL as a general component that can be specialized for different users. There may be situations where users have different needs regarding the DSL. For example "super users" having access to more features than ordinary users have. Or in creating a general DSL that can be tailored for each user according the the needs of that particular user. If you only need feature X, and the general DSL includes features X, Y and Z, then a specialized version only including feature X could be created.

Related to the specialization of a DSL is the ability to build new DSLs combining two or more existing ones. Each DSL can be seen as a building block providing functionality covering a piece of a domain. If a big domain is divided into sub-domains, each of the sub-domains could be implemented as separate DSLs. It would then be possible to build new DSLs only including support for the sub-domains needed. Two users working on separate parts of the domain could have their own tailored DSLs, with overlapping functionality where applicable.

Yet another form of customizing the DSL is by doing changes directly to the host language. Is it possible to remove, or "prune", features from the host language? For example if it is discovered that having access to conditional logic (like the `if/else` statements in Java) leads to more errors, then that conditional logic could be removed from the host language and customized logic added to the DSL instead. Similarly it is interesting to find ways to extend the host language. For example if you want to provide the users of the DSL with some kind of control structure not currently found in the host language, it would be useful to extend the host language with the needed control structure.

1.3.3 Using DSLs with Different Performance Requirements

There are many different ways a DSL could be used. For example as a library, as a standalone script being run from the command line, or in an interactive environment where commands are evaluated and executed as they are written. Do the different programming languages provide new ways to use DSLs over those provided by Java? A different aspect of using a DSL is that of error handling. How can exceptional situations be communicated to the user in an understandable way?

Different DSLs may have varying requirements when it comes to performance. For example a DSL for issuing stock

trades may have completely different performance requirements compared to a DSL created to support repeating numerical calculations. What host languages are suited for building high performance DSLs? Are there features in the languages that affect performance? How well do the different host languages facilitate building DSLs taking advantage of parallel processing and concurrency? There are many requirements to consider regarding performance.

1.4 Summary of Requirements and Report Structure

The rest of this report is structured around the requirements discussed in the previous section. After a description of the programming languages used and the examples implemented (chapter 2 «Programming Languages and Examples»), each of the requirements are treated in separate chapters. As described in the method section, each requirement is studied in terms of existing theory combined with examples. A summary is written for each chapter, concluding the results found related to the requirements covered by the chapter.

The following list summarizes the main requirements, and gives a pointer to what chapter they are covered in:

- **Syntax and Semantics** – What capabilities do the different programming languages provide in building concise syntax and semantics? Chapter 3 «Syntax and Semantics» looks at different host language features, and how those features can be used to create embedded DSLs with a concise and understandable syntax.
- **Customizing the DSL** – How can a DSL be tailored for a specific need? How can several DSLs be combined/reused into new ones? Chapter 4 «Creating Customizable DSLs» handles the design of such customizable DSLs.
- **Changing the Host Language** – How can unwanted features of the host language be deprecated/removed? How can the host language be extended to provide new functionality? Chapter 5 «Changing the Host Language» describes how the various host languages can be modified.
- **Use** – How can embedded DSLs be used? What is the level of tool support provided for the DSL? How are errors and exceptional situations handled in the different languages? Chapter 6 «Exploring Ways to Use Embedded DSLs» handles different ways to utilize and run the DSLs.
- **Performance** – How does the different languages facilitate building DSLs with high demands regarding throughput/performance? Chapter 7 «Performance» discusses various aspects regarding DSL performance.

Finally chapter 8 «Summary» provides a summary containing the main results and conclusion, as well as some suggestions to further work.

Chapter 2

Programming Languages and Examples

This chapter introduces the programming languages studied in the thesis, and the DSL examples used. As described in the introduction the main focus has been on capabilities provided by a set of different programming languages, and how these capabilities sustain the development of various types of embedded domain-specific languages. The examples description is divided into two sections. The first example section demonstrates the main example DSLs, implemented in Scala, while the second section introduces some other examples used.

2.1 JVM Programming Languages

The goal platform of this thesis is the JVM. One of the reasons for this is that the popularity of the Java programming language has made the JVM one of the most widespread multi-language platforms in the world. At the same time the Java language itself is getting old, and many other languages have been created (or ported) for the JVM. This thesis looks at how some of these languages provide various capabilities when building embedded DSLs. I have tried to look at languages that differ in many categories, as typing, paradigm or execution model. The languages studied are described and categorized in the next sections.

The following list summarizes other advantages in using the JVM as a platform:

- **Cross-platform distribution** – Code implemented in any of the languages can be run on any platform supporting JVM 1.5 or newer. The JVM is present on all major operating systems, and even on portable devices like mobile phones. There is no need to implement separate versions of DSLs for different platforms, as long as it runs on a standard JVM.

- **Stability and standards** – The JVM is a stable platform with good support for internationalization (full Unicode support) and localization (supports local languages, currencies and standards). A DSL can be used by users from different countries, localized to their native language and standards.

- **Access to libraries** – The JVM has a wide range of libraries available, many being Open Source. In many cases a DSL can be based on an existing Java framework.

- **High performance** – The JVM has been proven to be a high performance virtual machine, with a large number of tuning possibilities.

The main disadvantage using the JVM as a platform is in the limitations set by the currently supported bytecode, as it was developed targeting one language (Java). Many of the languages studied here have features that are not found in Java, and thus not being directly supported at the bytecode level. Examples of this is tail call optimization of recursive functions and dynamic dispatching of functions. An important aspect of the performance requirement will be to look into the performance of DSLs using these kind of features.

2.1.1 Categories of Languages

I have chosen to categorize the languages in three different ways:

- **Typing (static/dynamic)** – Static typing means that the majority of the type checking of the language is enforced during compile-time, as opposed to run-time. The opposite is dynamic typing which performs most of the type checking during run-time.
- **Paradigm (OO/FP)** – There are many different programming paradigms. I have looked at object-oriented and functional programming languages. Some of the languages (Ruby, Groovy and Scala) are multi-paradigm languages, combining object-orientation and functional programming techniques.
- **Execution (compiled/interpreted)** – On the JVM an interesting way to categorize programming languages is whether they are compiled to byte-code and run directly on the virtual machine, or if the language is interpreted and run by some extra layer on top of the JVM.

Table 2.1 shows the languages used in the thesis, and what categories they belong to. An `(x)` denotes partial support, a non-default configuration possibility, or that the language supports something similar. For example are both Groovy and Ruby said to support some notion of functional programming, although it is not their main focus. Scala and Clojure are both compiled languages that can be run in a REPL, a Read-Eval-Print-Loop compiling statements on the fly making it seem like they are interpreted. This gives both languages the possibility to run as scripts or directly in the console, as if they were interpreted (not compiled) languages.

	Age	Object-oriented	Functional	Static Typing	Dynamic Typing	Compiled	Interpreted
Java	1995	x	-	x	-	x	-
Scala	2003	x	x	x	-	x	(x)
Groovy	2003	x	(x)	-	x	(x)	x
JRuby	2001	x	(x)	-	x	(x)	x
Clojure	2007	-	x	-	x	x	(x)

Table 2.1: Programming languages comparison matrix.

Each of the languages are further introduced in the following sections.

2.1.2 Java

Java is naturally the most common programming language used on the JVM. It is a statically typed and compiled object-oriented language, created by the company Sun Microsystems in 1995 (version 1.0 released). In 2006/2007 most of Java was released as open source software, under the GNU General Public License (GPL). As such, Java is now considered a standard programming language independent of any particular company (although Sun still contributes much of the development, and are considered Java “evangelists”).

In this thesis Java is mainly used as a reference language, and as a baseline of what is possible on the JVM. Most of the work is focused on showing how other languages provide features useful when building DSLs, features not currently found in the Java language itself. So Java will be used for reference and comparison to the other languages. It is assumed that the reader has a certain knowledge about the Java language (because of its wide adoption in both academic and enterprise environments), and many Java concepts will only be described briefly or referred to without explanation.

Embedded DSLs can also be implemented in Java. A good background with examples can be found in the paper «Evolving an Embedded Domain-Specific Language in Java» [13], describing the development of the testing framework `jMock` as an embedded DSL in Java. The example shown in listing 2.1 on the next page is from [13], showing the DSL syntax of mock object expectations used in a mock `Offer` class. The techniques described in the paper can be seen as a sort of baseline for what is possible in Java. The majority of work done in this report is showing what can be achieved using features from other languages, going beyond what is possible in the Java language.

```
void testAcceptsOfferIfLowPrice() {
  offer = mock(Offer.class);
  offer.expects( once() )
       .method( "buy" )
       .with( eq(QUANTITY) )
       .will( returnValue(receipt) );
  ...
}
```

Listing 2.1: Java jMock example.

2.1.3 Scala

"Scala is a general purpose programming language designed to express common programming patterns in a concise, elegant, and type-safe way. It smoothly integrates features of object-oriented and functional languages, enabling Java and other programmers to be more productive. Code sizes are typically reduced by a factor of two to three when compared to an equivalent Java application." [35](Scala Website)

Scala is a relatively new language (first version released 2003 [35]) providing many new programming capabilities on the JVM. It is a statically typed and compiled language, just like Java. Scala is a multi-paradigm language, combining object-oriented and functional programming. This combination has been proved to support advanced embedding of DSLs, as shown by Hofer et al. [21]. The fact that the language compiles to bytecode and uses static typing makes it similar to Java in many ways. For these reasons Scala has been used as the main language in this thesis, and as example language when implementing the DSL examples. The other languages are used with smaller examples, demonstrating features not found in Scala (or Java).

On the object-oriented side, Scala provides many "new" features compared to Java. First of all it is more pure in its object-orientation, as all values are objects. There is no separation between primitive and object, as in Java (although the compiler might use primitives when producing bytecode for better efficiency). The language also supports advanced object-oriented features not found in Java, such as traits, singleton objects, class parameters, virtual types and higher-order generics (the details around these features will be explained later when used in examples).

Scala also has many features of functional programming included. This includes support for higher-order functions, pattern matching, type inference, implicit conversions, currying, partial functions, lazy data-structures and an advanced type system. The type system has been a showcase for what is possible in a static language on the JVM [28]. Many of the functional programming capabilities of Scala will be explained in detail later, with regards to the various DSL requirements.

A small Scala example is shown in listing 2.2 on the following page, showing some basics of the language. Functions are defined using the `def` keyword. Parenthesis are not mandatory when there are no parameters (as in the function `double`), and curly braces can be dropped if the function body consists of only one statement (as both functions in the example). Semicolons are usually only needed when having multiple statement on the same line. Using `val` defines an immutable value (like using `final` in Java), while `var` defines a variable (as is default in Java). The type of a value is written after the name. The reason for this is the support for type inference; the type does not have to be given if it is obvious to the compiler. A final point to note is that there is no explicit `return` statement. As in most functional languages the final statement in a function is considered the return value.

Scala and Java

Scala code (.scala files) compiles to Java bytecode (.class files) through a special compiler (`scalac`). The bytecode can then be run on the JVM as if it was created from Java source (.java files). In addition Scala provides a number of library classes that need to be included as a jar file. This relationship between Scala and Java is illustrated in figure 2.1 on the next page. Scala does not separate classes implemented in Java or Scala. Classes can be imported and used directly, with no special syntax or handling.

```
// Example trait
trait MathTrait {
  def addNumbers(x: Int, y: Int) = x + y
}

// Example class
class MathDoubler(var num: Int) extends MathTrait {
  def double = addNumbers(num, num)
}

// Example of use
val m = new MathDoubler(32)
println(m.double) // prints 64
m.num = 64
println(m.double) // prints 128
```

Listing 2.2: Scala introductory example.

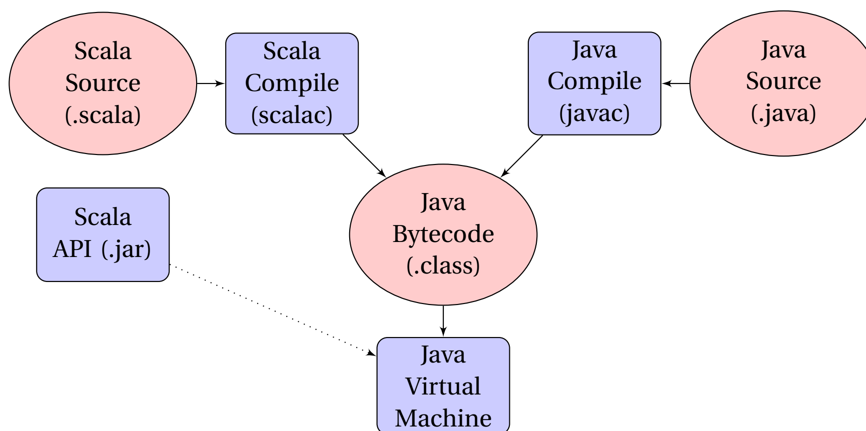


Figure 2.1: Visual representation of the relationship between Scala and Java.

2.1.4 Groovy

Groovy is a dynamically typed and interpreted (or optionally compiled) object-oriented language. The first beta-versions of the language were available in 2003, but the first official release (version 1.0) was not until 2007. I have included Groovy in this thesis as it provides an interesting dynamically typed alternative to the Java language. It has some syntax-capabilities making it well suited for building embedded DSLs.

An example of a simple object-oriented "Hello World!" program is shown in listing 2.3. The keyword `def` is used for both variables and functions. Strings support insertion of values directly, as with the `name` variable in this example. Comments in Groovy are the same as for Java (and Scala).

```
class Greet {
  def name

  Greet(who) { name = who }

  def salute() { println "Hello \$name!" }
}

// Groovy comments are the same as in Java
new Greet('world').salute()
```

Listing 2.3: Groovy "hello world" example.

Like Scala, Groovy supports higher-order functions. The code in listing 2.4 on the next page shows an example. An

anonymous function is created with the `it * it` expression, and assigned to the variable `square`. This can then be used as a regular function to calculate the square value of numbers. Arrays are created using square brackets. In the example an array of four integers is created, and the `collect` function is run passing the previously created `square` function as parameter. This loops through the array, calling the given function for each element, and returning the result as a new array.

```
// Define anonymous function 'square'
square = { it * it }
// Example of use:
nine = square(3)

// Create an array and pass the 'square' function as parameter to the 'collect' function
squareNumbers = [1, 2, 3, 4].collect(square)
// Prints [1, 4, 9, 16]
println(squareNumbers)
```

Listing 2.4: Groovy anonymous function example.

Groovy and Java

Groovy is tightly integrated with Java. In most cases, an existing Java code file can be renamed to `.groovy` and run directly. This is due to the fact that Groovy supports most of the native Java syntax, as well as all the "new" dynamic style syntax. All Java classes can be imported and used directly in Groovy. Most Groovy functionality uses the existing functionality in Java, with its own extensions added.

2.1.5 Ruby/JRuby

Ruby is a dynamically typed and interpreted object-oriented language, first released in 1995 (same year as first version of Java). JRuby is a Java implementation of the Ruby language on the JVM, thus acting as an interpreter for Ruby on the JVM. JRuby was first released in 2001, but not used much until support for Rails was added in 2006. The latest versions of JRuby also support some notions of just-in-time compilation, as well as regular ahead-of-time compilation.

The main reason for my inclusion of JRuby in this thesis is the "Ruby on Rails" framework [20], a Ruby DSL running on the JVM with the help of the JRuby interpreter. In listing 2.5 is a Ruby implementation of the same "hello world" program that was shown in Groovy in the previous section. In Ruby blocks are ended with the `end` keyword, instead of being surrounded by curly braces (as in Java, Scala and Groovy). Functions are defined using the `def` keyword. Instance variables are created using the `@` character, as in the `@name` variable in the example. Comments are lines starting with a # character.

```
class Greet
  @name

  def initialize(name)
    @name = name
  end

  def salute()
    puts "Hello #@name!"
  end
end

# A Ruby comment
Greet.new("world").salute()
```

Listing 2.5: Ruby "hello world" example.

JRuby and Java

JRuby can use classes from the standard Java API directly using the `include Java` statement. It can also import Java classes directly from jar files. So most Java libraries and APIs can be reused when working with JRuby on the JVM. An example of this is shown in listing 2.6 on the next page, creating a small Java Swing application.

```
# Special include to make standard Java API available
include Java

frame = javax.swing.JFrame.new("Main Window")
frame.getContentPane.add(javax.swing.JLabel("Press button:"))
frame.getContentPane.add(javax.swing.JButton("OK"))
frame.setDefaultCloseOperation(javax.swing.JFrame::EXIT_ON_CLOSE)
frame.pack
frame.setVisible(true)
```

Listing 2.6: Ruby Java integration example.

2.1.6 Clojure

Clojure is a dynamically typed and compiled functional language. It is a modern dialect of the Lisp programming language compiling to Java bytecode. It is the only language not supporting object-orientation directly, and also the youngest and possibly the least mature language included in my work. Clojure was first released in 2007.

The main reason I have looked at Clojure is the amount of embedded DSL papers written using functional programming languages as host language ([1], [23], [24] and [25]), making Clojure an interesting language with respect to embedding DSLs on the JVM. In listing 2.7 is an example of the "hello world" function in Clojure, as well as an example implementation of the Fibonacci function (a more common "hello world" in the functional world). All the code is grouped in lists, given by the parenthesises. Every statement in Clojure is a list on the form `(<fn> <arg1> <arg2> ...)`, where `<fn>` is the name of a function followed by any number of arguments.

```
; A kind of hello world function
(def hello (fn [] "Hello world"))

; The fibonacci function
(defn fib [n]
  (if (<= n 1)
    1
    (+ (fib (- n 1)) (fib (- n 2)))
  )
)
```

Listing 2.7: Clojure Fibonacci example.

Clojure has a big collection of features not found in any of the other languages, many of which will be explained in more detail where relevant. In Lisp (and thus in Clojure), the most important fact is the simplicity of the basics in the language. There are very few special forms requiring specific syntax. All code is organized in lists, meaning that code is made up of data and vise versa, making Clojure a "homoiconic" programming language [19] («Programming Clojure», a good book for learning more about Clojure). The macro system utilizes this fact, making it possible to seamlessly mix code and data to provide new control structures. Clojure macros will be used with examples several places in this report, together with other features such as multimethods and built-in immutable data structures.

Clojure and Java

Clojure is the language studied that has the least in common with Java, with regards to syntax and appearance. Many people not being used to the syntax of Lisp may find Clojure difficult to read, with all the parenthesis being a central part of the structure. Figure 2.2 on the next page shows a basic example with a variable, function and some output, implemented side by side in Clojure and Java. In many ways Clojure is much more consistent in its syntax. While Java has special keywords and forms for defining variables, functions, operator use and more, Clojure simply does everything the same way. Notice how the definition of the variable in Clojure (`(def x 5)`) is on the same list-form as the definition of the function (`(defn halfOrSome ...)`), and that the comparison operator used in the if statement in Java is also just a regular function in the same form as the definitions in Clojure. The simple list-structure and functional style in Clojure gives the code a more consistent look than that of Java.

```
; Define x as variable
(def x 5)

; Define halfOrSome as function
(defn halfOrSame [n]
  (if (> n 2)
    (/ n 2)
    n))



; Print result of calling function with variable
(println (halfOrSame x))
```

(a) Clojure

```
// Define x as variable
int x = 5;

// Define halfOrSome as function
public static int halfOrSame(int n) {
  if (n > 2)
    return n / 2;
  else
    return n;
}

// Print result of calling function with variable
System.out.println(halfOrSame(x);
```

(b) Java

Figure 2.2: Example comparing syntax consistency of Clojure and Java side by side.

Clojure integrates with Java using some special macros like `import` and `new`, and the dot operator, as shown in the examples given in listing 2.8. Comments are defined as lines starting with semicolon. In the example a system property is extracted using the static Java method `getProperty` of the `System` class. Notice how the function name always comes first, even in the case of calling the `toUpperCase` method on a Java string. Finally it is shown how the `java.util.Date` class can be imported, instantiated and used.

```
; Get a system property
(System/getProperty "java.vm.version")
; Call "toUpperCase" on the String "clojure"
(.toUpperCase "clojure")
; Import java.util.Date, make a new instance and print the class name
(import java.util.Date)
(new Date) ; New instance of Date
(.getName Date) ; Call "getName" on the Date class
```

Listing 2.8: Clojure – Java integration.

In addition all Clojure data structures implement standard Java interfaces, making it easy to run code implemented in Clojure from Java. The good interoperability between Clojure and Java is a bit surprising looking at the differences in syntax and programming paradigm, but is a powerful addition to the Lisp world. Combining the macros of a Lisp language with all the available libraries on the Java platform seems to be a good idea.

2.2 Examples

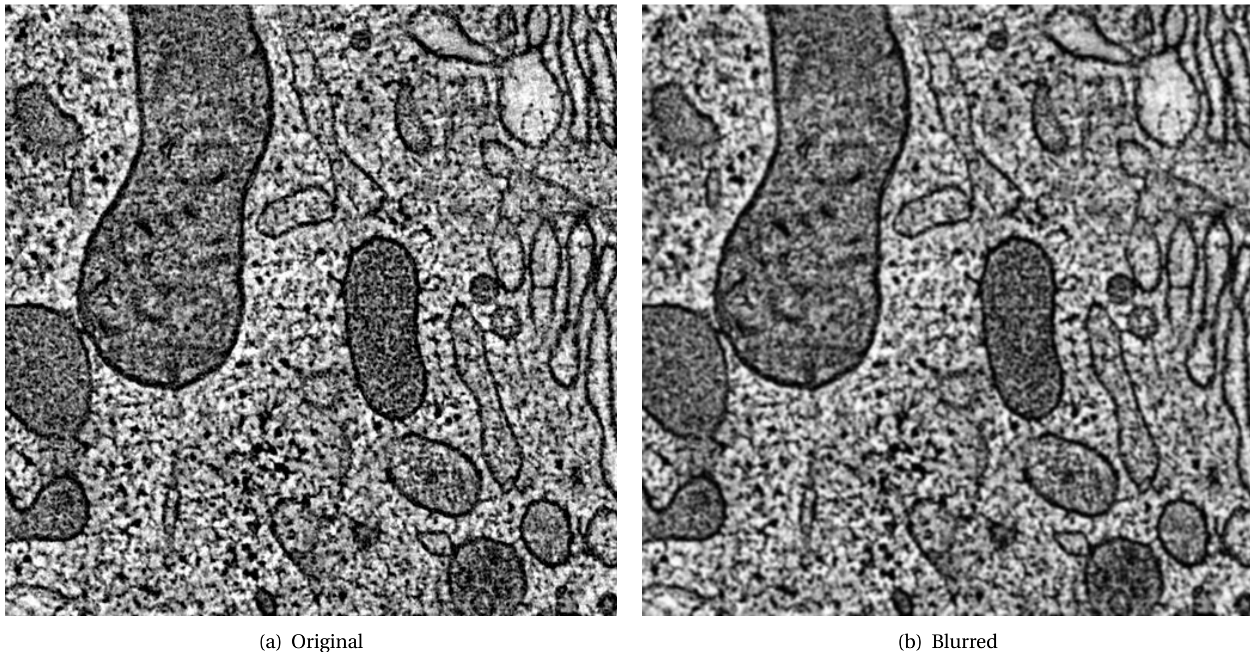
This section describes the main examples implemented and researched in the thesis. The DSLs are described, and details about implementation and key features given. Some source code being central in the implementation is shown in the text. Full source code for all the examples can be downloaded from my repositories at GitHub, full instructions are given in Appendix A on page 89.

2.2.1 DSL for Image Processing

Description

Image Processing is an area involving series of computationally heavy operations performed on images [17]. Practitioners in the area have typically more mathematical than computer science oriented background. This leads to the need for DSLs where complicated and computationally heavy operations can be performed with a simple and straightforward syntax, hiding much of the programming language specific, letting the users focus on their domain.

A major actor in the art of Image Processing is the company MathWorks™with their product MATLAB®. They provide the user with a DSL for doing advanced Image Processing operations. The goal of this example is to create a similar DSL (with a limited amount of functionality) running on a JVM. The DSL was implemented in Scala, as



(a) Original (b) Blurred

Figure 2.3: Example image before and after blurring with average filter.

the main language studied. It is important to note that the focus has been on creating a best possible DSL example, not on doing anything new in the area of image processing.

As an example consider the operation of blurring an image using an average filter mask. This is a basic operation, documented in [17], but serves good as an example of using the DSLs. The result of the operation is shown in figure 2.3, original and blurred images side by side. The process is illustrated in figure 2.4 on the next page, showing how each pixel in the new image is an average of an area of pixels in the old image. This is a good example to use, as it involves potentially many operations (depending on the image size) such that performance considerations also must be taken.

The MATLAB® code to perform the operation could typically look like that of listing 2.9. An image is read from file, blurred with the `medfilt2` function, and the result saved back as a new file.

```
img = imread('cell.jpg');
% 3x3 neighborhood default
imgAvg = medfilt2(img);
imwrite(imgAvg, 'cell_avg.jpg');
```

Listing 2.9: MATLAB® image processing example.

A Scala-solution to the same problem is shown in listing 2.10. The same image is loaded, and then blurred using a “structuring element” [17]. The result is then written back to a new file. The rest of this section describes how this was implemented, with regards to syntax and class/object composition. The DSL also implements some other basic image processing operations like erode, dilate, open and close [17] that are not shown here.

```
// Imports...
val img = loadImageCP("/cell.jpg")
val se = StrEl(Square, 3)
val imgAvg = img.avg(se)
saveImage(imgAvg, "/cell_avg.jpg")
```

Listing 2.10: Scala image processing example.

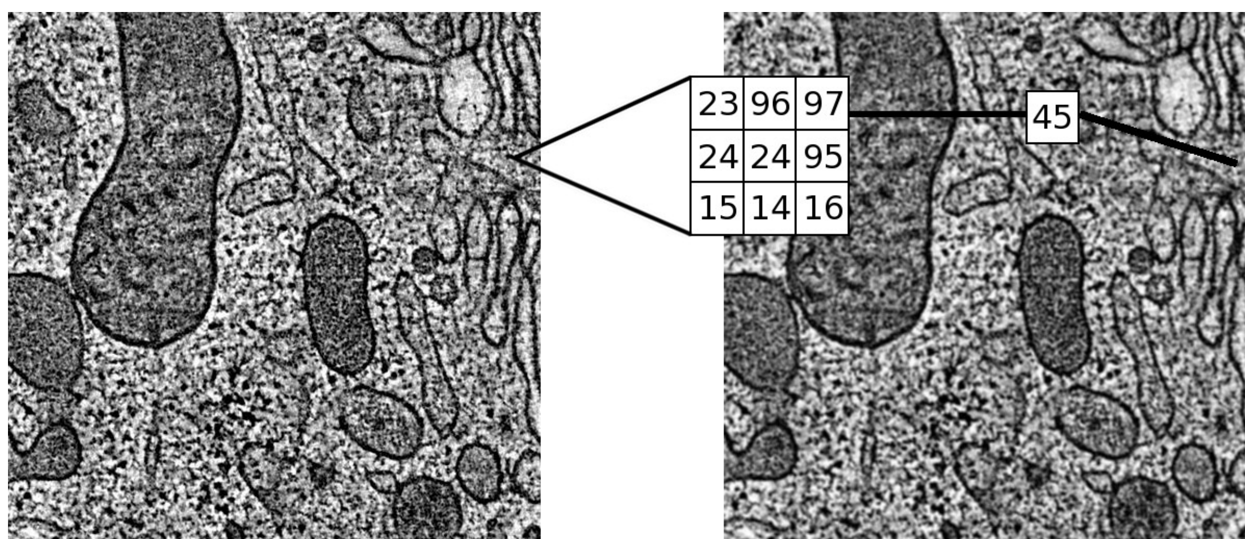


Figure 2.4: Visualising the process of blurring an image using average filtering with a 3x3 neighbourhood.

Syntax

This section briefly describes the syntax and implementation of the example. Some Scala features are used that will be further explained in the next chapter, and are only described here with short footnotes.

The functions to load and save the image, as well as the `Square` definition of structuring element, are all member functions in classes `SImageIO` and `StrElType`. These can be imported with the `._` notation in Scala, and are then available directly.[4] An example is shown in listing 2.11, where the comments show how the statements would look without using the static member import.

```
import io.SImageIO._
import structs.StrElType._

// SImageIO.loadImageCP("/cell.jpg")
val img = loadImageCP("/cell.jpg")
...
saveImage(imgAvg, "/cell_avg.jpg")

// StrElType.Square
val se = StrEl(Square, 3)
```

Listing 2.11: Scala image processing syntax example.

To create the structuring element, the `StrEl` trait has a "companion object"[5] with an `apply` method (described in section 3.5.1 on page 33). This gives a compact syntax, combined with the import of the available structuring element types as shown above. An example is shown in listing 2.12.

```
// Definition of companion object StrEl
object StrEl {
  import StrElType._
  def apply(t: StrElType, num: Int) = Array2DStrEl(t, num)
}

// Example of use - from previous example
val se = StrEl(Square, 3)
```

Listing 2.12: Scala companion object example.

[4]The Scala member import is similar to the static member import found in Java, making static methods in a class directly available.

[5]A companion object is a singleton `object` having the same name as an associated `trait` or `class` [32].

Using traits[6], we are able to compose image objects with the operations we want. The operations are implemented in separate traits which are mixed in with the image on creation. This is using selftype annotations[7] to specify that the trait can only be mixed in with classes of type `GrayScaleImage`. See listing 2.13 for a small example, where the trait `Standard` implements a function `avg` that will be available on instances of `GrayScaleImage` having the trait mixed in (as is done in the `Image` object in the example).

```
// Trait defining standard operation 'avg'
trait Standard { this: GrayScaleImage =>
  def avg(se: StrEl[Int]) = { ... }
}

// Image class constructing objects from specific classes and traits
object Image {
  import operations.{Morphology, Standard}

  def apply(d: Matrix[Int]) = {
    new GrayScaleImage(d) with Standard with Morphology
  }
}

// Example of use
val imgAvg = img.avg(se)
```

Listing 2.13: Scala image processing traits.

Performance and Parallel Computing

With images larger than a certain size, the operations performed in the image analysis are bound to be heavy in terms of the number of computations that need to be executed. This opens up a demand for exploiting the capabilities found in modern computers with regard to parallelization and concurrency. As such the image processing DSL provides the basis for studying the performance of Scala running on the JVM. Parallelizing the image processing DSL will be explored in section 7.3 on page 73, in chapter 7.

2.2.2 Statistics DSL

Statistic processing is an important aspect in many numerical calculation environments. For example in image analysis, statistical models are used for pattern recognition (ie. finding text or other figures in images). In this respect it is interesting to have a DSL for statistical operations that can be used together with the image processing DSL to create a more complete DSL environment for image analysis.

A minimal DSL for statistics has been created in Scala for this work. It only has a basic data structure, with a few rudimentary operations. The main reason for creating this little DSL is to provide a basis when working with DSL composition (see chapter 4 on page 45).

This whole DSL consist of a single `DataSet` class, implementing the basic operations `average`, `minValue` and `maxValue`. There is also a `DataSet` companion object acting as a factory for building data-sets. The source code is given in listing 2.14 on the facing page. The `DataSet` class takes as parameter an array of integers, and defines the three basic functions mentioned above. The implementation of the functions uses anonymous functions (_ + _) as parameters to the `reduceLeft` function on the array. This is a typical example of combining the object-oriented and functional programming features of Scala. The factory object uses a variable number of integers (denoted with the type `Int*`) to build the array needed for the dataset.

Using the statistics DSL is typically done by creating a data-set from a series of integers, and then calling the wanted operations. An example calculating and printing (to the standard console) the average value of a standard 6-sided dice (numbers 1-6) is given in listing 2.15 on the next page.

[6] Traits are an object-oriented mechanism allowing mixin-composition with classes. This can be seen as a form of basic multiple inheritance.

[7] Selftype annotation is a way to give a trait a specific type. A trait with a given selftype can only be mixed in with objects of that type. [33]

```
class DataSet(private val intArr: Array[Int]) {
   def average = intArr.reduceLeft(_ + _) / intArr.size.toDouble

   def minValue = intArr.reduceLeft(_ min _)

   def maxValue = intArr.reduceLeft(_ max _)
}

object DataSet {
   def apply(ints: Int*) = new DataSet(Array(ints: _*))
}
```

Listing 2.14: Minimal statistics DSL implemented in Scala.

```
val dice = DataSet(1, 2, 3, 4, 5, 6)

println("Dice average: " + dice.average)
```

Listing 2.15: Scala statistics DSL usage.

2.2.3 Charting DSL

Another feature that can be useful when working with numeric calculations is the ability to create charts displaying results, statistics or just a more visual representation of the data. As with the statistics DSL described above charting is important when working with image analysis, typically creating histograms of images, visualizing statistical distributions and many other uses.

The DSL created for charting is also a minimal one, as with the statistics DSL. It provides the third and last important piece when building a full DSL environment for image analysis. The DSL is implemented in Scala (as the two DSLs described in previous sections), and implements the charting functionality using the Java library JFreeChart.[8] It shows how an existing Java library can be directly used as implementation for a DSL implemented in the other languages.

An example creating a basic pie chart is shown in listing 2.16, resulting in the chart shown in figure 2.5 on the next page. The whole example consists of two functions, `createPieChart` and `pieData`, creating the chart with a variable number of data values.

```
createPieChart("Pie Chart",
   pieData(
      ("One", 43.2D),
      ("Two", 10D),
      ("Three", 27.5D),
      ("Four", 17.5D),
      ("Five", 11D),
      ("Six", 19.3D)))
```

Listing 2.16: Scala charting DSL – create a basic pie chart.

2.3 Other Examples Used

This section describes some other DSLs used as examples. They have not been implemented as part of the work, but are used as examples as they are popular Open Source frameworks being used in many enterprise projects. They also exercise some exciting features in the host languages to obtain some of the features that make them suited for the domains they provide solutions for.

[8] `http://www.jfree.org/jfreechart` – Open source (LGPL licensed) charting library implemented in Java.

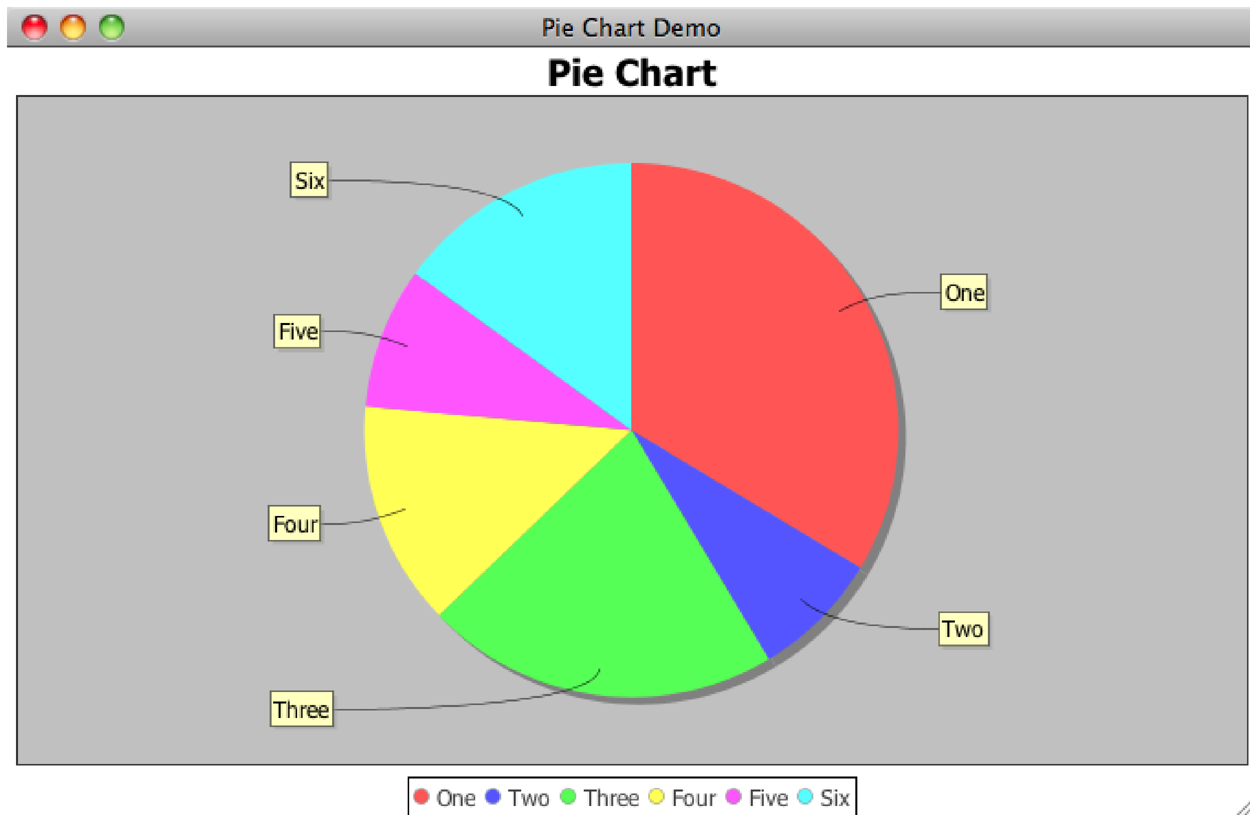


Figure 2.5: Example pie chart created with the JFreeChart based charting DSL.

2.3.1 Ruby on Rails

Ruby on Rails is a framework for web development. It is implemented as an embedded DSL in Ruby. The main Rails feature focused on in this thesis is the `ActiveRecord` classes, providing functionality for querying databases. The `ActiveRecord` can be seen as a separate DSL on its own. An example of using `ActiveRecord` is shown in listing 2.17 on the facing page (from [20]). The relationship between tables in the database is directly mapped to Ruby classes, using a specific syntax. In the example the `Client` class will be mapped to a "client" table in the database, and the `Address` class to an "address" table and so on. The framework takes care of producing the SQL code needed to bind data to objects when classes are instantiated, and makes sure that foreign keys refer to the correct tables based on the relationship mappings like `has_one` and `has_many`.

2.3.2 Hibernate

Hibernate is a so called object-relational-mapping (ORM) framework implemented in Java. It provides much the same functionality as the `ActiveRecord` mechanism described in the previous section, but implemented in Java using annotations. In listing 2.18 on the next page, the `Client` class from the Ruby on Rails example given in 2.17 on the facing page is implemented in Java using Hibernate.

2.3.3 Testing Frameworks

Several frameworks for automated unit testing are good examples of embedded DSLs on the JVM. In this thesis jMock[9] and ScalaTest[10] have been used as examples. jMock is implemented in Java, and is a framework providing functionality for mocking dependencies so that classes can be tested in isolation. ScalaTest is implemented in

[9] jMock – A Lightweight Mock Object Library for Java: `http://www.jmock.org/`
[10] ScalaTest – Open-source testing framework for Scala: `http://www.scalatest.org/`

```
class Client < ActiveRecord::Base
  has_one :address
  has_one :mailing_address
  has_many :orders
  has_and_belongs_to_many :roles
end

class Address < ActiveRecord::Base
  belongs_to :client
end

class MailingAddress < Address
end

class Order < ActiveRecord::Base
  belongs_to :client, :counter_cache => true
end

class Role < ActiveRecord::Base
  has_and_belongs_to_many :clients
end
```

Listing 2.17: Example using the Ruby on Rails `ActiveRecord` database querying classes.

```
@Entity
public class Client

  @ManyToOne
  public Address getAddress();

  @ManyToOne
  public Address getMailingAddress();

  @OneToMany
  public List<Order> getOrders();

  @ManyToMany
  public List<Role> getRoles();
}
```

Listing 2.18: Java Hibernate example

Scala, and is a unit testing framework that facilitates several different styles of testing. Both jMock and ScalaTest are excellent examples of frameworks taking full advantage of their respective host languages. An example using jMock was shown in listing 2.1 on page 17.

Chapter 3

Syntax and Semantics

"The syntax of a language defines its surface form." [14]

It is the set of rules that define the combinations of symbols that are considered to be correctly structured programs in that language. For an embedded DSL its syntax describes special ways to use the host language. Or in other words what defines the code as an embedded DSL is the way the syntax constructs of the host language are used. The semantics will typically follow those of the host language, plus any additional semantics implemented by code. This chapter describes various programming language concepts that can be useful when designing an embedded DSL. These concepts form the building blocks that one can choose from when building a DSL. Choice of host language determines what features will be available.

3.1 Function Chaining/Nesting

As mentioned in section 1.3.1 on page 12, Fowler and Evans have used the term fluent interfaces to describe libraries that can be used in a certain way. A key functionality when creating such fluent interfaces in Java is the ability to chain/nest functions together.

One of the simplest ways to create a DSL-like syntax is to allow chaining several function calls together. As an example consider the following Java-code (from [12]), where the construction of a `HardDrive` object is shown with and without method chaining:

```
// Construct with setter methods
HardDrive hd = new HardDrive();
hd.setCapacity(150);
hd.setExternal(true);
hd.setSpeed(7200);

// Construct with method chaining
HardDrive hd = new HardDrive().capacity(150).external().speed(7200);
```

Listing 3.1: Java function chaining example.

The chaining of methods can also be combined with method nesting, grouping methods that belong together (also from [12]). An example of this is shown in listing 3.2 on the next page.

In [13], this technique is described in detail. In the testing framework `jMock` a carefully designed set of interfaces and factory builder objects allow method chaining maintaining the wanted order of method calls. This ordering is a nice example of how custom semantics can be built into the DSL syntax. The example code in listing 3.3 on the following page shows how this can be done using traits in Scala (or interfaces in Java). The key point is to let each method return an instance of the next possible type. In the example the method of `StartSyntax` return the type `BodySyntax` which again has methods returning type `EndSyntax`. The traits can all be implemented by the same class, returning `this` in each method. When the syntax is instantiated as type `StartSyntax` the specific

```
computer(
  processor(
    cores(2),
    Processor.Type.i386
  ),
  disk(
    size(150)
  ),
  disk(
    size(75),
    speed(7200),
    Disk.Interface.SATA
  )
);
```

Listing 3.2: Java method chaining and nesting.

ordering of method calls will be maintained by the compiler. The `BodySyntax` is made optional by extending the `EndSyntax` type.

```
// Traits defining a syntax
trait StartSyntax {
  def start: BodySyntax
}

trait BodySyntax extends EndSyntax {
  def doSomething: EndSyntax
  def doSomethingElse: EndSyntax
}

trait EndSyntax {
  def end: Unit
}

class Syntax extends StartSyntax with BodySyntax {
  ... // Method implementations returning this
}

// Instantiate Syntax as type StartSyntax
val syntax: StartSyntax = new Syntax

// Examples of ok use, body is optional
syntax.start.doSomeThing.end
syntax.start.end

// Example NOT ok - compile error, wrong order of statements:
syntax.start.end.doSomeThing
```

Listing 3.3: Scala traits providing DSL order.

Function chaining is available in most programming languages. In functional programming it is a fundamental concept, while in object-oriented languages more a design choice (as in the example above). In [24], a technique using monads in Haskell is used to create functions that can be combined in various ways to create embedded DSLs. Similar examples could be created in Clojure and Scala to run on the JVM.

3.2 Static Methods/Singleton Objects

In object-oriented languages functions are nested inside classes. In order to be able to call functions without instantiating objects we have static methods, that are associated directly with the class instance and not with objects created from the class. The example in listing 3.4 on the next page shows how to sort a list of strings in Java, using the static method `sort` in the `Collections` class.

```
// Create a list of strings
List<String> names = ...

// Sort the list using the static sort method in the Collections class
Collections.sort(names);
```

Listing 3.4: Java static sort example.

A different approach to static methods is the singleton objects found in Scala. Instead of having methods defined as static there is a notion of a singleton object, as in the following example defining sorting similar to the Java-example above:

```
// Define ListSorter object
object ListSorter {
  def sort(list: List) { ... }
}

// Use object
var names = List("name1", "name2")
ListSorter.sort(names)
```

Listing 3.5: Scala singleton object sort example.

Static methods may play a central role in an embedded DSL, especially combined with member imports (next section). They are typically used for object creation through patterns like "Factory method"[11] and "Builder"[12], or other utility operations like the sorting shown in the examples above. In the image processing DSL (described in section 2.2.1 on page 21), singleton objects are used to construct objects. An example is shown in listing 3.6.

```
// Using singleton object:
val squareElement = StrEl(Square, 3) // 3x3 square element
val lineElement = StrEl(HLine, 5) // 5x1 line element

// Without singleton object, using new:
val squareElement = new StrEl(Square, 3)
val lineElement = new StrEl(HLine, 5)
```

Listing 3.6: Scala image processing singleton objects.

3.3 Member Imports

With member imports it is possible to make members of a class available directly in the code that is being implemented. Member imports is a useful mechanism in object-oriented languages for creating concise syntax, and thus in the embedding of DSLs. Java supports importing static members of classes, both methods and constants. Using static member import the Java sorting example from previous section can be implemented as the code in listing 3.7, omitting the name of the `Collections` class where the `sort` method is used.

```
import static java.util.Collections.*; // Import all static members of the Collections class
...

// Create a list of strings
List<String> names = ...

// Sort the list
sort(names);
```

Listing 3.7: Java static member import example.

[11] The factory method is an object-oriented pattern that handles the creation of objects without specifying the exact class of the object [15].

[12] The builder pattern is an object-oriented pattern that assists the creation of variations of objects through a series of abstract method calls [15].

In Scala member imports are less restricted than in Java. They can be used anywhere in the code (not just at the start of the file), and are not restricted to static members.

```
class Fruit(val name: String, val color: String) // Define class Fruit with two members

val apple = new Fruit("apple", "red") // Create apple as instance of Fruit

import apple._ // Import all members from object apple

println(name + " is " + color) // Access members 'name' and 'color' directly
```

Listing 3.8: Scala member import example.

The image processing DSL uses a number of member imports to enable a concise syntax when working with the DSL, allowing the user of the DSL to use the method/type names directly (such as the `loadImageFile` method and the `Square` type in the example below):

```
import simage.io.SImageIO._
import simage.structs._
import simage.structs.StrElType._

val img = loadImageFile("cell.jpg") // Would be SImageIO.loadImageFile without member import
val se = StrEl(Square, 3) // Would be ...StrElType.Square

saveImage(img.avg(se), "cell_avg.jpg") // Would be SImageIO.saveImage without member imports
```

Listing 3.9: Scala image processing member imports.

3.4 Method Names and Operator Notation

When constructing an embedded DSL we like to make the syntax reflect the actual domain as much as possible. Having flexible ways to name and call methods can be a useful property of a host language. Scala has two features regarding method names and syntax that are interesting with regards to DSL development [6]:

- Arbitrary method names - Scala methods can have special characters as names. As such Scala has full support for operator overloading, as operators are just regular methods with special names. For example +, != and <= are all valid method names in Scala.
- Operator notation - Methods in Scala can be called without the dot and parentheses, just as regular operators are used in Java.

The example in listing 3.10 shows how the operator notation is used together with the + method name in a simple matrix DSL (being an important part of the larger image processing DSL shown in section 2.2.1 on page 21). Notice how there is no difference between the method with operator name and the method with a "regular" name in the way they can be used.

```
abstract class Matrix {
  // Method names can be operators
  def +(other: Matrix) = add(other)
  // Regular method name
  def add(other: Matrix): Matrix // Abstract method
}

// Usage example - regular notation
val m3 = m1.+(m2)
val m3 = m1.add(m2)
// Operator notation
val m3 = m1 + m2
val m3 = m1 add m2
```

Listing 3.10: Scala operator notation function example.

Combining the method naming in Scala with method chaining can give “fluent interfaces” with even more concise syntax than what was shown in previous sections. The example in listing 3.11 is from the website of ScalaTest, a testing framework supporting Behaviour Driven Development[13] in a fluent style. Notice how operator notation is used in all method calls related to the DSL testing syntax. Other parts of the example will be explained in later sections.

```
import org.scalatest.FlatSpec
import org.scalatest.matchers.ShouldMatchers

class StackSpec extends FlatSpec with ShouldMatchers {

  "A Stack" should "pop values in last-in-first-out order" in {
    val stack = new Stack[Int]
    stack.push(1)
    stack.push(2)
    stack.pop() should equal (2)
    stack.pop() should equal (1)
  }

  it should "throw NoSuchElementException if an empty stack is popped" in {
    val emptyStack = new Stack[String]
    evaluating { emptyStack.pop() } should produce [NoSuchElementException]
  }
}
```

Listing 3.11: Scala ScalaTest example.

The dynamic language Ruby has a mechanism similar to that found in Scala. Here you are allowed to leave the parentheses surrounding the parameter(s) out, but not the dot in front of the method name. In addition it is a common idiom in Ruby to append method names with a question mark if the method returns a boolean value. Some examples of use are shown in listing 3.12. In many cases, leaving the parenthesis out renders more human-readable code when using Ruby.

```
name = "John"

# Method without parameters
name.empty?()
name.empty?

# Method with one parameter
name.include?("e")
name.include? "e"
```

Listing 3.12: Ruby parameter example.

3.5 Special Syntax Constructs

Many programming languages have special constructs that are made specifically to enable the writing of concise statements. Typically constructs that only offer a different syntax that will be replaced by the compiler at early phases of compilation. The next sub-sections describe some examples of such mechanisms.

3.5.1 The “magic” `apply` Method

Scala uses the `apply` method to let classes and objects define functionality that appears to be native in the language. You can leave the method name out and simply write syntax that seems to pass parameters directly to an object. In listing 3.13 on the next page the `apply` method is used to index the `Matrix` class as if it was a native feature. The user of the code does not have to write the name of the `apply` method, as it is inserted by the compiler in the background.

[13] BDD is a software development technique first introduced in [30]. It aims at writing automated tests in a natural language, and has been implemented as embedded DSLs in a number of different languages.

```
// Class with generic apply method signature
class Matrix[T] {
  def apply(row: Int, col: Int): T = ...
}

// Example of use - get number at position 2, 4 from some matrix of integers
val intMatrix = ...
val intVal = intMatrix(2, 4) // Replaced by compiler: intMatrix.apply(2, 4)
```

Listing 3.13: Scala `apply` method example.

The `apply` method can also be used for object construction without using the `new` keyword, and without exposing the actual class being used to create the object. This makes it possible to create code according to the factory method pattern [15], as in listing 3.14 (simplified version of Matrix structure from the image processing DSL). Notice how the user simply refers directly to the `Matrix` object, without any knowledge about the complicated structure of classes that implements the matrix data structure.

```
abstract class Array2DMatrix[T](elements: Array[T]) extends Matrix[T] {
  ... // Some abstract specialization of the Matrix class
}

class IntArray2DMatrix(elements: Array[Int]) extends Array2DMatrix[Int](elements) {
  ... // Concrete implementation of an int matrix
}

// Singleton object being used to create instances of IntArray2DMatrix
object Matrix {
  def apply(ints: Int*) = new IntArray2DMatrix(Array(elements: _*))
}

// Example use, create matrix from a number of integers - hiding details about the classes shown above
val matrix = Matrix(1, 2, 3, 4, 5, 6, 7, 8)
```

Listing 3.14: Scala `apply` method factory example.

As we can see from the above example, combining singleton objects with the `apply` method can effectively hide a complex object-oriented underlying structure. The user of the DSL can focus on only the details that are relevant (as the list of numbers making up the matrix in the example above, implementation details being irrelevant).

3.5.2 Query Syntax

In C# 3.0 a DSL called LINQ (Language Integrated Query[14]) was introduced. It is used to natively query data in the language. The implementation of the DSL relies on a number of language features (type inference, anonymous types, object initializer and lambda expressions – see sections to follow) in addition to the query syntax itself. An example is shown in listing 3.15 on the next page. Notice how the datastructure is queried using a specific syntax. The bottom part of the example shows how the compiler will interpret the query syntax, as a combination of different nested method calls.

In Scala we have `for comprehensions` that can be viewed as a query syntax much like that of LINQ. Complex expressions using the mechanism will be rewritten to a combination of the functions `filter`, `flatMap` and `map` by the compiler. An example is shown in listing 3.16 on the facing page. We can see that the conditional if statement in the for-comprehension is converted to a `filter` function using an anonymous function (to be explained in a later section). The yielding of the value is converted to a `map` function, with yet another anonymous function.

Daniel Spiewak has written a paper investigating the possibility to extend the for comprehensions in Scala to support LINQ-style database queries [34]. Both mechanisms rely on monadic behaviour (concept from functional programming), where complex statements can be constructed by combining a set of basic functions (as shown in

[14]LINQ website: `http://msdn.microsoft.com/en-us/library/bb308959.aspx`

```
// Query syntax:
var results =
  from c in SomeCollection
  let x = someValue * 2
  where c.SomeProperty < x
  select new {c.SomeProperty, c.OtherProperty};

// Resulting rewritten syntax:
var results =
  SomeCollection.Where(
    c => c.SomeProperty < (SomeValue * 2)
  ).Select( c => new {c.SomeProperty, c.OtherProperty} );
```

Listing 3.15: C# LINQ example.

```
// "Query syntax" - for comprehension:
val results = for {
  name <- listOfNames
  if (name startsWith "E")
} yield { name toUpperCase }

// Resulting rewritten syntax:
val results = listOfNames.filter(_ startsWith "E").map(_ toUpperCase)
```

Listing 3.16: Scala `for comprehension` example.

the examples above). These mechanisms are good additions directly in a DSL. Alternatively a custom query syntax could be implemented for a specific DSL, using the techniques described here and in the next sections.

Clojure supports a mechanism that is similar to the query languages described above, namely relational algebra on the `Set` data-structures [19]. As with the Scala for-comprehensions the whole system relies on a small number of functions (set union, set difference, rename, selection, projection and cross product), that can be combined in various ways. Code listing 3.17 shows a small example of use (taken from [19]), extracting data from sets containing information about compositions, composers and nations.

```
(project
  (join
    (select #(= (:name %) "Requiem") compositions)
    composers)
  [:country])
```

Listing 3.17: Clojure relational algebra example.

3.6 Type Inference

With type inference the compiler will figure out the type of a value or function, so there is no need to explicitly specify it. This is useful when writing embedded DSLs, to give a less verbose and more concise syntax. Some examples were given in the previous section about query syntax.

The Scala compiler will try to infer the types used. So if the type is obvious there is no need to specify it, as shown in the following example where various types are inferred:

```
val i = 42 // Type Int is inferred
val d = i + 2.1 // Type Double is inferred
val iBy2 = (i: Int) => i * 2 // Return value Int is inferred from int parameter
val dBy2 = (i: Double) => i * 2 // Return value Double is inferred from int parameter
```

Listing 3.18: Scala basic type inference example.

This makes the code similar to that of a dynamically typed language. An advantage when working with embedded DSLs in dynamic languages is that there is no need to explicitly write the types. With Ruby or Groovy the first two definitions in the example above could simply be implemented as in listing 3.19 (not even needing the `val` used in Scala). Type errors would, however, not be discovered until runtime. With Scala the type errors will still be captured at compile-time, so type inference is a way to write more concise code without loosing the safety-net provided by the compiler.

```
i = 42
d = i + 2.1
```

Listing 3.19: Ruby/Groovy variable definitions.

In the image processing DSL there is extensive use of the type inference mechanism. As we can see from the example in listing 3.20 there is no explicit naming of the types involved, as opposed to the bottom part where the same code is given with the types. As the `avg` function is specified in the `Standard` trait being used with the `GrayScaleImage` the full type “GrayScaleImage with Standard” must be given, as well as the `Int` type parameter to the `StrEl`. The example also displays the part type inference plays in hiding the implementation details from the user of the DSL. With type inference the user does not need to be aware of the types returned (what traits are used and so forth).

```
// Actual implementation, using type inference
val img = loadImageFile("cell.jpg")
val se = StrEl(Square, 3)
val blurredImg = img.avg(se)

// Without using type inference the code would look like this:
val img: GrayScaleImage with Standard = loadImageFile("cell.jpg")
val se: StrEl[Int] = StrEl(Square, 3)
val blurredImg: Image = img.avg(se)
```

Listing 3.20: Scala type inference example.

3.7 Implicit Type Conversion

Implicit type conversions are used to allow the compiler to automatically convert one type to another where possible. From a DSL syntax perspective this is used to "add" new methods to existing types. This can be built-in types or user-defined types. In the following Scala example an implicit conversion is used to convert from `Int` to `MyInt`, and thus allowing the `doubleIt` method to be called directly on the integer value:

```
// Wrapper class for int
class MyInt(val i: Int) {
  def doubleIt = i * 2 // Method
}

// Implicit conversion
implicit def fromInt(i: Int) = new MyInt(i)

// Example of use - converts and Int to MyInt and calls method doubleIt
val ten = 5.doubleIt
```

Listing 3.21: Scala implicit type conversion example.

Implicit conversions are commonly used when creating embedded DSLs in Scala. A good example is found in the ScalaTest example shown in section 3.4 on page 32, where the `String` type is used as if it had defined the methods `should` and `in`. The value returned from the `pop` operation on the stack being tested is treated the same way. The relevant parts of the example are shown in the bottom part of listing 3.22 on the next page. What makes this possible is implicit conversions, shown in the upper part of the same code listing. For the `String` class the `should` method is defined in a class called `StringShouldWrapper`, which is implicitly available since there is defined a conversion from `String`. The value popped off the stack is converted to an `AnyShouldMatcher` with a similar implicit conversion, including a generic type parameter.

```
// Method signatures
implicit override def convertToStringShouldWrapper(o: String): StringShouldWrapper

implicit def convertToAnyShouldWrapper[T](o: T): AnyShouldWrapper[T]

// Usage example showing relevant parts
...
"A Stack" should "pop values in last-in-first-out order" in {
...
  stack.pop() should equal (2)
  stack.pop() should equal (1)
}

it should "throw NoSuchElementException if an empty stack is popped" in {
...
```

Listing 3.22: ScalaTest example using implicit type conversions.

In dynamic languages a mechanism that can give much the same results is Monkey Patching described in section 3.13 on page 43. Implicit conversions are also useful when combining DSLs. This will be further described in chapter 4 on page 45.

3.8 By-name Parameters

In Scala, parameters to methods can be defined as by-value or by-name. By-value is the default, and means that the parameter is evaluated before the method is invoked. By-name is the contrast to by-value, and means that the evaluation of the parameter happens every time the method implementation refers to the parameter.

An example where this feature is used is shown in listing 3.23. The method `times` in the class `DoIntTimes` has a by-name parameter `body` of type `Unit` (means no value, much the same as `null` in Java). The class wraps an integer value `n`, which is used in the `times` method to evaluate the by-name parameter n times. In the example the class is combined with an implicit conversion from `Int`, to provide the handy DSL syntax «2 times ... ».

```
// Class implementing function with a by-name body parameter
class DoIntTimes(val n: Int) {
  def times(body: => Unit) {
    for(i <- 0 until n) body
  }
}

// Implicit conversion from integers to the class defined above
implicit def intToDoIntTimes(n: Int) = new DoIntTimes(n)

// Example of use
2 times {
  println("printed twice!")
}
```

Listing 3.23: Scala example combining implicit conversion with by-name parameter.

3.9 Lambda Expressions/Anonymous Functions

Anonymous functions, or lambda expressions, can be used as parameters for higher-order functions.[15] They are useful to make concise code, and avoiding duplication. For example in the Scala for comprehensions and the LINQ statements shown in section 3.5.2 on page 34, anonymous functions play a role in allowing filtering (if-statements).

To show how anonymous functions work consider the following example. The goal is to filter all even numbers

[15]Higher-order functions are functions that can take other functions as input parameters or return a function as output value, or functions that can be stored in variables.

from a list, typically done with the modulo operator (%). Writing this code in Java, which does not support anonymous functions, would result in something similar to the example in listing 3.24 using a `for-loop` combined with an `if-statement`.

```
List<Integer> numbers = ... // Make instance of list of integers
List<Integer> evenNumbers = new ArrayList<Integer>();

for(int number : numbers) {
  if(number % 2 == 0) evenNumbers.add(number);
}
```

Listing 3.24: Java simple for-loop/if example

In a language supporting anonymous functions, the `List` class could have a `filter` method taking a function as parameter which is used to decide what values should be returned in a new list. In Scala, the example in listing 3.24 could be implemented as follows. Notice how the concise the syntax for defining the anonymous function is. The underscore is used as an anonymous parameter, since the filter method expects a function taking one parameter this can be anonymized:

```
val numbers = ... // Make instance of list of integers
val evenNumbers = numbers.filter(_ % 2 == 0)
```

Listing 3.25: Scala basic anonymous function example

It is quite clear that anonymous functions make the code much more to-the-point and concise. In the Scala example we could leave out both the loop and the if statement, only focusing on the actual operation being performed (the modulo 2 equal to 0). If several different similar operations were to be performed, the example would be even clearer. In Java the loop would have to be repeated several times, or at least several if-statements would have to be included in the loop, while in Scala (or other languages supporting anonymous functions) the operations would occur in an ordered series of filter-statements.

In my Matrix DSL I have used higher-order functions to create general-purpose functions to perform operations spanning a given structuring element. The signature looks like this:

```
def seOp(se: StrEl[Int], op: (Seq[T]) => T): Matrix[T]
```

It may look cryptic at first, but is really quite simple. The first parameter (`se`) is a structuring element (typically spanning a small 3x3 section of the matrix), and the second (`op`) is a function converting a sequence of values to one value. The structuring element is passed over the matrix as a window, and the operation is run for all values covered by the element. The yielded value is set as a value in a new matrix, which is the final return value of the whole function. In the image processing DSL based on the Matrix DSL this functionality is used to implement a number of operations, using anonymous functions. Some examples are shown in listing 3.26 on the next page. Notice how the only thing differentiating the functions is the actual implementation of the anonymous function setting the rule for how two numbers should be handled.

These implementations should look familiar to anyone with a knowledge of image processing, which is the target user domain of the DSLs. Users of the image processing DSL would never have to see the anonymous functions, but could use the function `blur`, `erode` and `dilate` directly, as shown in listing 3.27.

```
val image = ... // Obtain image instance
val se = StrEl(Square, 3) // Create 3x3 structuring element
val blurredImage = image.blur(se)
val erodedImage = image.erode(se)
```

Listing 3.27: Scala image processing functions implemented using anonymous functions.

Anonymous functions are also available in Ruby, Groovy and Clojure. They are often referred to/intermixed with closures[16] in Ruby and Groovy, and as lambda expressions in Clojure. Support for anonymous functions has also

[16]Closures are actually anonymous functions binding a variable defined in a surrounding scope, but the difference is not important to the usages described here (or most other places, really).

```
val matrix: Matrix[Int] // Internal image representation as a matrix

// Use average anonymous function to create blur effect
def blur(se: StrEl[Int]) = {
  Image(matrix.seOp(se, (seq) => seq.reduceLeft(_ + _) / seq.size))
}

// Use minimizing anonymous function to create morphological erosion effect
def erode(se: StrEl[Int]) = {
  Image(matrix.seOp(se, (seq) => seq.reduceLeft(_ min _)))
}

// Use maximizing anonymous function to create morphological dilation effect
def dilate(se: StrEl[Int]) = {
  Image(matrix.seOp(se, (seq) => seq.reduceLeft(_ max _)))
}
```

Listing 3.26: Scala anonymous function example.

been discussed for the next version of Java.[17]

3.10 Anonymous Types

Anonymous types allow the construction of types without a name, based on fields. Combined with type inference this is a central feature in the implementation of LINQ (as described in an earlier section). For a DSL doing custom querying this may be a useful feature.

Scala also supports anonymous types. An example of using anonymous types together with type inference:

```
val person = new { val name = "John"; val age = 26 }

println(person.name + " is " + person.age + " years old..")
```

Listing 3.28: Scala anonymous types example.

In LINQ this is used to allow selection of fields from a database or other data structure:

```
var results =
  ... // Some query
  select new {c.SomeProperty, c.OtherProperty};
```

Listing 3.29: C# LINQ example using anonymous types.

In the functional world of Clojure, all data-structures are basically anonymous types. They are defined as maps of key-value pairs. The “person” type shown in the Scala example above can be defined similarly in Clojure:

```
(def person {:name "john" :age 25})

(println (str (person :name) " is " (person :age) " years old.."))
```

Listing 3.30: Clojure data structure example.

3.11 Object Initializer

Object initializers provide the ability to give values to an object while instantiating it. In C# this can look like the following, and is also used to select objects of a given class instead of anonymous types from LINQ:

[17] Several different proposals for closures/anonymous functions in Java 7 have been discussed. The main effort is now lead by Mark Reinhold, as stated on his blog (`http://cr.openjdk.java.net/~mr/lambda/straw-man/`), and is documented as “Project Lambda” at the OpenJDK: `http://openjdk.java.net/projects/lambda/`.

```
val results =
  ... // Some query
  select new Person { Name = c.SomeProperty, Age = c.OtherProperty }
```

Listing 3.31: C# LINQ object initializer example.

Scala also supports a form for object initializers, as in the following example:

```
class Person {
  var name = ""
  var age = 0
}

val person = new Person { name = "John"; age = 26 }
```

Listing 3.32: Scala object initializer example

In embedded DSLs this can be useful when building query-like syntaxes like that shown for LINQ. If the properties used to construct the objects vary, it would mean a lot of extra code if special constructors need to be made for every combination of properties. Just the two properties `name` and `age` used in the above examples would result in 4 different constructors (none, one x 2 or both properties given).

3.12 Metaprogramming

Metaprogramming is the writing of programs that write or manipulate other programs (or themselves) as their data, or that do part of the work at compile time that would otherwise be done at runtime. In terms of embedded DSL creation, metaprogramming can be a useful mechanism. There are big differences between the studied programming paradigms in how they provide metaprogramming, how it is used and what you can do with it. The next sub-sections describe some of these differences with examples.

3.12.1 Metaprogramming in Static Languages

Java has some mechanisms associated with metaprogramming:

- **Annotations** – Annotations were introduced to Java with version 5 (JDK 1.5). They are used to add metadata about a program. These data are not part of the program itself, and can be used both at compile-time/deployment-time and at runtime. Deployment-time annotations can be used for code generation, and can be used when embedding DSLs.

- **Reflection** – Reflection is a way to program with concepts such as `class` and `method` in Java, and as such is considered a form of metaprogramming. Extensive use of reflection can have a negative effect on the performance of a Java application.

- **Aspect Oriented Programming (AOP)** – AOP is a mechanism that can be used for metaprogramming. As shown by Huang et al. in [22], AOP and metaprogramming can be useful when designing DSLs.

An example of an embedded DSL in Java relying heavily on annotations is the Hibernate framework, providing an object-relational mapping between Java domain objects and database tables. The example in listing 3.33 on the next page shows Hibernate in work mapping the classes `Person` and `Address` to database tables. A person has one address, but many persons can live on the same address. This relationship is mapped using annotations. With these annotations in place, Hibernate will generate the mapping (SQL-code) needed to extract correct data from the database.

Scala has no special mechanism for metaprogramming beyond those offered in Java (Scala also supports annotations and reflection), and has not been considered in this section.

```
@Entity
public class Person {
  @Column(name="person_name", length=100)
  public String getName() { ... }

  @ManyToOne
  public Address getAddress() { ... }
}

@Entity
public class Address {
  ...
  @OneToMany
  public List<Person> getPersons() { ... }
}

// Example of use
Address adr = ... // obtain some address object
// The getPersons() method will actually run some SQL in the background fetching correct persons
for(Person person : adr.getPersons()) { ... }
```

Listing 3.33: Java metaprogramming with annotations.

3.12.2 Dynamic Metaprogramming

Dynamically typed languages are able to use metaprogramming in a more dynamic way, to add or alter method definitions at runtime. The dynamic typing allows referring to names of functions that don't yet exist at compile-time, while a static language would end up with a compiler error in the same situation. Both Groovy and Ruby have support for this kind of dynamic metaprogramming.

The example in listing 3.34 shows how the mechanism can be used in Groovy. In this case a new method is added to the Java `String` class[18], and the static `random` method of the `Math` class is changed to a not-so-random implementation useful for unit testing or other environments where you want to have full control of returned values.

```
// Add a "shout" method to the Java String class
String.metaClass.shout = {
  println delegate.toUpperCase()
}
"hello".shout() // prints "HELLO"

// Handy when you want to do controlled unit testing?
Math.metaClass.static.random = {
  return 42
}
```

Listing 3.34: Groovy metaprogramming example.

Ruby supports dynamic metaprogramming in a manner similar to that shown for Groovy. A typical example of a DSL utilizing metaprogramming in Ruby is the Rails framework and the `ActiveRecord` class. Here metaprogramming is used to specify relationships between the classes, with respect to database relationships. The SQL code needed to fetch related objects will be generated as new methods. In the example in listing 3.35 on the next page this technique is shown for the classes `Team` and `Player`. The methods `has_many` and `belongs_to` are implemented in a manner similar to the Groovy examples above, adding methods `team` and `players` with database extraction logic to the classes in use. A simplified example of how the `belongs_to` method is implemented using dynamic metaprogramming in Ruby is shown in listing 3.36 on the following page, running some SQL and creating a collection of associated objects dynamically. An important principle to make this work is that strict naming conventions are followed in the database. Database names can be overridden in the code with arguments to the methods, but that would make the example less "clean".

[18] Note that the `String` class is final in Java, but still modifiable with dynamic metaprogramming in Groovy.

```
class Team < ActiveRecord::Base
  has_many :players
end

class Player < ActiveRecord::Base
  belongs_to :team
end

# Now a player will have a 'team' method returning the team
player.team

# And a team will have a 'players' method returning a collection of players
team.players.each { |team| ... }
```

Listing 3.35: Ruby on Rails example showing dynamic metaprogramming.

```
class ActiveRecord
  ...
  def self.has_many(arg)
    name = arg.to_s
    send :define_method, name do
      sql = "select * from #{name} where #{self.class.name.downcase!}_id = #{self.object_id}"
      # Run sql, and return new objects based on sql result
      ...
    end
  end
  ...
end
```

Listing 3.36: Ruby dynamic metaprogramming example.

This dynamic style metaprogramming is one of the biggest advantages that dynamic languages have when it comes to designing embedded DSLs. Using annotations or other metaprogramming techniques could be used in Java or other static languages to generate code similar to the one generated by the rails framework. However, the client would not know about the methods until after compiling. So programming in this fashion in a static language would require some intermediate step of explicit code-generation or compiling before calling the generated methods. I think this is one of the big differentiators when it comes to certain kind of DSLs and programming language paradigms. A DSL like Rails could not have been implemented in a static language.

3.12.3 Functional Metaprogramming

In Clojure, the only “pure” functional language used in this thesis, there is no big separation between code and data. It is a “homoiconic” language, meaning that Clojure code is composed of Clojure data [19]. As such the whole concept of metaprogramming is an integrated part of the language, and not necessarily seen as a separate concept (as in the other languages). The macro mechanism in Clojure is the key to this.

Clojure has a programmatic macro system which allows the compiler to be extended by user code.[19] This means that we can write code that produces code, which falls under the definition of metaprogramming given earlier. The Clojure macros are processed in two steps. First the macro expands (executes), and substitutes the result back into the program. This is called *macro expansion time*. Then it continues with the normal *compile time* [19]. Clojure macros provide similar DSL capabilities to those achieved with “C++ Template Metaprogramming”, described by Czarnecki et al. in [4].

The code in listing 3.37 on the next page shows a Clojure macro example (from [19]). The `bench` macro is a simple benchmarking tool to time how long an expression takes to complete, returning a map containing both the result of the expression and the time it took to execute (in nanoseconds). The `let` expression binds the start-time to the `start#` name, and the results of the expression given to the `result#` name. The symbols ` and ~ are used to quote

[19] `http://clojure.org/macros`

and un-quote symbols in the macro expansion. It finishes off by returning the result together with the computed time difference. It may not look that impressive, but it is a powerful mechanism that you will not find anything similar to in any of the other languages. In fact, much of the Clojure library is implemented as macros, even the `defmacro` definition used to define macros is itself a macro.

```
(defmacro bench [expr]
  `(let [start# (System/nanoTime)
         result# ~expr]
     {:result result# :elapsed (- (System/nanoTime) start#)}))

; Example usage
(bench (+ 2 3))
; Output: {:result 5, :elapsed 27000}
```

Listing 3.37: Simple Clojure macro example.

3.13 Open Classes/Monkey Patching

A monkey patch is a way to extend or modify the runtime code of dynamic languages without altering the original source code. In DSL creation this can be used in much the same way as implicit conversions, by adding new functionality to built-in or other types. In the example in listing 3.38 code we re-implement with open classes/monkey patching the same example shown in section 3.7 on page 36 with implicit type conversions, giving the integer class a new method `doubleIt`. Note that the `Integer` class already exists, and is simply modified with this code. If it did not exist it would be defined here, as the syntax is the same for defining new classes and modifying existing ones.

```
# Add method doubleIt to class Integer
class Integer
  def doubleIt
    self * 2
  end
end

# Example of use
ten = 5.doubleIt
```

Listing 3.38: Ruby example showing “monkey patching” on open classes.

In a DSL monkey patching will typically be used to add functionality to base classes, similar to what was shown with implicit conversions in listing 3.22 on page 37. With the implicit conversion in Scala the `String` type was converted to `StringShouldWrapper`, making it seem like it had a `should` method. With monkey patching in Ruby it is possible to add the method directly to the `String` class. A small example, suggesting how the signature would look with an example of how it could be used, is shown in listing 3.39.

```
# Add method should to class String
class String
  def should(arg)
    ...
  end
end

# Example of use
"A stack" should ...
```

Listing 3.39: Ruby `should` method added to String.

3.14 Dynamic Dispatch – Method Missing

Dynamic programming languages use dynamic dispatch to determine what code to run when a message is received. This means that errors related to missing methods are postponed until runtime. Some languages provide

ways to handle these errors programatically, that can be utilized in DSL creation.

In Ruby there is a notion of custom dispatch behaviour that can be controlled by the programmer. The base class, that all other classes extend, has a `method_missing` method that is called by the dispatcher as a last resort when no other method is found to respond to a message. The default behaviour is to throw a `NoMethodError` (similar to what would happen in Java or other languages), but this can be overridden by extending classes.

To see how this can be used in a DSL we can look at a simplified version of the `ActiveRecord` class from the `Rails` framework, as shown in listing 3.40. It maps objects to database tables and allows dynamic querying through the DSL syntax. In the example the method named `find_by_username` is mapped to data extraction functionality through the `method_missing` implementation. Notice how a regular expression is used to extract the different parts of the original method name as separate values. These values can then be used to build and run some kind of dynamic SQL expression.

```
class ActiveRecord
  # ...
  def self.method_missing(method_id, *arguments)
    if match = /find_(all_by|by)_([_a-zA-Z]\w*)/.match(method_id.to_s)
      # ... extract column names from match, generate and run SQL expression + map and return results
    end
  end
end

# Define class 'Person' as extension of 'ActiveRecord'
class Person < ActiveRecord::Base
end

# Query the database for person with given user name
person = Person.find_by_username "eivindw"
```

Listing 3.40: Ruby on Rails `ActiveRecord` example.

This dynamic dispatch mechanism is another example where dynamic languages provide unique possibilities. With a static language the missing method would result in a compiler error, and hence not be possible to implement.

3.15 Summary

Modern statically typed languages like Scala offer some useful features for building embedded DSLs, combining concepts from object-oriented and functional programming languages. Advanced type systems offering type interference, implicit conversions and anonymous types provide powerful mechanisms when constructing DSLs, removing much of the verbosity often associated with statically typed languages.

When comparing various classes of programming languages the biggest difference with regards to DSL syntax seems to be that between static and dynamic languages. Even with feature rich static programming languages like Scala there are a series of concepts from dynamic languages which does not have a natural counterpart in the statically typed domain. Most notably are the metaprogramming features found in many dynamic languages and the dynamic dispatch mechanism allowing custom handling of missing methods.

Chapter 4

Creating Customizable DSLs

This chapter shows how DSLs can be customized for specific needs. It is divided in two main parts; the first describes techniques used to customize a general DSL to more specialized versions, and the second showing how to combine several DSLs to create new ones. The main example used is the image processing DSL. It is shown how it can be specialized to a specific version, and also how it can be combined with the other example DSLs (statistics and charting) to create a more complete environment for image analysis.

4.1 Specializing a General DSL

This section describes several ways to create tailored versions of a specific DSL. A key to achieve this is in making the DSLs modularly composed. The first sub-section discusses mixin-composition in the different programming languages studied, and show how the image processing DSL uses traits in Scala to build modules of functionality that can be combined in different ways. Then it is shown how type specialization and abstract types can be used to build general DSLs. The image processing DSL also uses these techniques to make it easier to extend into more specialized versions. The final sub-section describes some functional programming techniques that can be used to build modularly implemented DSLs.

4.1.1 Mixin-composition – Polymorphic Abilities

Scala has support for family polymorphism [31] and mixin composition [33]. These are features that support polymorphic embedding of DSLs [21]. This allows building DSLs that can have several different interpretations as reusable components. It can also be used to effectively combine different DSLs into new ones. The paper "Polymorphic Embedding of DSLs" [21] describes this in detail with examples.

A use of traits can be to split functionality in several traits and create classes/objects with only the methods that we need. In the example given in listing 4.1 on the next page the `Matrix` class does not have any methods, but instances of the class can be mixed in with traits containing various methods. Notice how the three instances created at the end of the example all have different methods available, because of the way they mix in different traits.

Mixin composition is also available in Groovy and Ruby. In Groovy there is a general mixin meta-class function available, making it possible to combine several classes into one. An example of using this mechanism in Groovy is shown in listing 4.2 on the following page. Because of the dynamic typing the `name` variable of the `Base` class is available in the mixin-classes, as if they were subclassing the base class. This is different from the Scala traits example, where selftype annotations has to be used if the traits need to access any members of the classes they are mixed in with [33].

In Ruby there is a `module` concept that provides dynamic mixin composition functionality similar to that shown above in Groovy. Modules are a way to logically group classes, functions or other definitions belonging together, as an advanced form of the packages found in Java and Scala. Modules cannot be instantiated like classes, but can be

```
class Matrix {} // No methods

trait AvgMtx { this: Matrix =>
  def avg(): Matrix = { ... }
}

trait MorphMtx { this: Matrix =>
  def erode(): Matrix = { ... }
  def dilate(): Matrix = { ... }
}

// Matrix with only avg() method
val m1 = new Matrix with AvgMtx
// Matrix with erode() and dilate() methods
val m2 = new Matrix with MorphMtx
// Matrix with "all" methods
val m3 = new Matrix with AvgMtx with MorphMtx
```

Listing 4.1: Scala traits example.

```
class Base {
  String name;
}

class StdOp {
  def std() {
    println("std() running.." + name);
  }
}

class ExtraOp {
  def extra() {
    println("extra() running.." + name);
  }
}

Base.class.mixin(StdOp, ExtraOp);

base = new Base(name: 'Test!');
base.std(); // Base has a std() function
base.extra(); // ..and an extra() function
```

Listing 4.2: Groovy mixing composition example.

mixed in with a new or existing class to add functionality. Given in listing 4.3 on the next page is the same example as was shown for Groovy above, implemented using modules in Ruby.

Using the modular composition techniques it is possible to extract parts of a DSL as a new one, by only including the traits (or mixins/modules) wanted. An optimizing compiler could also utilize this composition to only include the parts of a DSL that are actually used, thus optimizing the size of the bytecode produced. This would be useful if compiling the DSL for a computer with a limited amount of resources (memory and storage), for example a mobile or other small device having support for running Java applications.

Aspect Oriented Programming in Java

Aspect Oriented Programming (AOP) is a separate programming paradigm providing support for mixin-composition (aspects are closely related to mixins) and more. It is not supported by Java directly, but several frameworks exist adding AOP support to the language. Some examples of frameworks are AspectJ[20], JBossAOP[21] and Qi4j.[22] The

[20] http://www.eclipse.org/aspectj/
[21] http://www.jboss.org/jbossaop
[22] http://www.qi4j.org/

```
class Base
  @name

  def initialize(name)
    @name = name
  end
end

module StdOp
  def std
    puts "std() running.." + @name
  end
end

module ExtraOp
  def extra
    puts "extra() running.." + @name
  end
end

class MyBase < Base
  include StdOp, ExtraOp
end

base = MyBase.new("Test!")
base.std() # MyBase has a std() function
base.extra() # ..and an extra() function
```

Listing 4.3: Ruby mixin composition using modules.

paper «Domain-Specific Languages and Program Generation with Meta-AspectJ» [22] describes "Meta-AspectJ", a language for generating AspectJ programs using code templates, and how it is used to construct a DSL. It is clear that using AOP is a good way to achieve many of the same benefits in Java, that were shown for other languages using mixin-composition. AOP is even more powerful than mixin-composition, as it allows a much more fine-grained mechanism for aspect weaving than that of pure polymorphism. Composition can be made simply by means of names, not using any form of inheritance at all. This is possible due to AOP being implemented with dynamic proxies[23] or similar reflection/metaprogramming techniques in Java. Although not studied in detail here, as it is not really an integrated part of the Java language (and thus outside the scope of this thesis), it is worth being aware of the capabilities that can be achieved using AOP.

4.1.2 Type Specialization and Abstract Types

Another way that a DSL can be created to allow specialization is by using virtual types (abstract type members) and/or type parametrization (generics) of classes. Scala supports higher-order genericity [29] and abstract type members. Scala uses the erasure model of generics, just like Java does [8]. This means that no information about type arguments is maintained at runtime, with the exception of arrays which are handled specially (as they also are in Java). In this section it is demonstrated how these features are used to make the image processing DSL more general, allowing specializations in a number of ways. The main reason for using these features in the image processing DSL is to ease the process of extending with new specializations.

The `Matrix` trait used in the image processing DSL uses type parametrization to denote the type of the values in the matrix. A matrix of integers will be defined as `Matrix[Int]` and a matrix of floating point numbers as `Matrix[Double]`. The matrix trait is extended by an array specific implementation, in the form of an abstract class `Array2DMatrix` also using the same type parameter as the matrix trait. Finally concrete classes `IntArray2DMatrix` and `DoubleArray2DMatrix` extend the `Array2DMatrix` providing integer- and double specific implementations. Together with a companion object acting as a matrix factory, these definitions are shown in listing 4.4 on the following page. A user of this DSL can create matrices specific for integer or double values without knowing the concrete implementations actually in use. This is shown at the end of listing 4.4. Another benefit

[23] `http://java.sun.com/javase/6/docs/api/java/lang/reflect/Proxy.html`

in using type parameters is that users can extend the matrix data structure with support for new types simply by extending the `Matrix` trait with whatever type they need.

```
trait Matrix[T] {
  ... // Implementation of general matrix interface
}

abstract class Array2DMatrix[T] extends Matrix[T] {
  ... // 2D-array specialization of the general matrix interface
}

class IntArray2DMatrix extends Array2DMatrix[Int] {
  ... // Integer specific implementation of the abstract 2D-array-matrix type
}

class DoubleArray2DMatrix extends Array2DMatrix[Double] {
  ... // Double specific implementation of the abstract 2D-array-matrix type
}

// Companion object acting as factory for matrix objects
object Matrix {
  def apply(nCols: Int, els: Array[Int]) = new IntArray2DMatrix(nCols, els)

  def apply(nCols: Int, els: Array[Double]) = new DoubleArray2DMatrix(nCols, els)
}

// Examples of use - construct one matrix of each specific type based on the type of the arguments
val intMtx = Matrix(2, Array(1, 2, 3, 4))
val doubleMtx = Matrix(2, Array(1.1, 2.2, 3.3, 4.4))
```

Listing 4.4: Matrix example using type parameters.

While type parameters are exposed at the signature of a trait or class, virtual types are kept inside the body. This gives similar abilities to those found with type parameters, but may feel different for the users. In the image processing DSL, a virtual type `DataType` is used in the `Image` trait to give the type representing the data structure of the image. This is shown in listing 4.5. In the specific implementations of images, the datatype is specified as a type suited to represent the kind of image. Gray scale images (images only using different values of gray, resulting in a “black and white” look) can be represented with one integer for each pixel, giving the gray-level value, as the `GrayScaleImage` class. Notice how the generic `Matrix` trait is used to make the datatype a matrix of integers. Color images can be represented as the `RGBImage`, with the datatype being a matrix of triple-integers (one integer value for each of the colors Red Green and Blue).

```
trait Image {
  type DataType
  ... // Implementation of general image interface
}

class GrayScaleImage(data: Matrix[Int]) extends Image {
  type DataType = Matrix[Int]
  ... // Implementation of gray scale image, as a matrix of integer values
}

class RGBImage(data: Matrix[(Int, Int, Int)] extends Image {
  type DataType = Matrix[(Int, Int, Int)]
  ... // Implementation of RGB image, as a matrix of triple integer values (representing the three colors)
}
```

Listing 4.5: Image example using virtual type.

4.1.3 Functional Composition

Obtaining modular composition is typically dependent on some kind of dynamic dispatch mechanism, as the polymorphism described with the use of traits in the previous section. Clojure, being a functional language, supports a different mechanism to obtain similar effects. This is called *multimethods* [19], and is basically a kind of

dynamic dispatch where you specify a function to perform the dispatch rather than only the type as is done in polymorphism. Instead of the type itself providing some implementation of an interface, the whole dispatch is done outside using the type as a parameter. A basic example is shown in listing 4.6. The `my-print` function uses the `class` function, which returns the class of an object, to dispatch on the argument. So the method can be called on objects of type `String` or `Number` (super-type for numeric types in Java). This example does basically the same as object-oriented languages do with polymorphism, it dispatches based on the type of the argument.

```
(defmulti my-print class)

(defmethod my-print String [s]
  (.write *out* s))

(defmethod my-print Number [n]
  (.write *out* (.toString n)))
```

Listing 4.6: Clojure multi-method example

Multimethods, however, are not limited to only dispatching on the type of an object. They can dispatch on any function, and can use more than one parameter when dispatching. In [19] a part of the Clojure core API using multimethods is explained, namely the *inspector* mechanism.[24] This can be seen as a kind of DSL to provide quick views of data organized in a tree-like structure. The interesting part of this is that the data-structures used can be anything you want. The only thing that needs to be done to add support for a new type in the structure is to define with multimethods how the type should respond to the tree-operations `is-leaf`, `get-child` and `get-child-count`.

A challenge when dealing with dispatch mechanisms that are so flexible is how conflicts should be resolved. This is the same problem found when dealing with multiple inheritance in object-oriented languages. If two or more implementations are found in the dispatch process, which one should be chosen? The Scala traits solve this with linerization [32], where the order the traits are given in decides which one “wins”. With Clojure multimethods the user can specify resolution to such conflicts programatically using a macro called `prefer-method`, that explicitly specifies which implementation should be used when several qualify based on the dispatch.

The multimethods mechanism is a way to customize the way functions are dynamically dispatched in a DSL. It is a complicated mechanism that probably should be used with care, but the possibilities it provides are not found in any of the other languages studied in this thesis.

4.2 Combining DSLs

This section describes how to combine two DSLs into a new one, mixing the functionality of the two original DSLs.

4.2.1 Implicit Conversion

Implicit type conversions were described in section 3.7 on page 36, in terms of DSL syntax. In this section we will look at how implicits can be used to combine functionality from two different DSLs. The concept is demonstrated with an example.

The following code shows the use of the simple DSL for statistics. Typically some kind of data-set is created with numeric values (in this example only integer values for simplicity). The data-set implements the methods `average`, `minValue` and `maxValue`. In the most basic form it can be used directly to compute the average, minimum and maximum values of a series of integers:

```
val ds = DataSet(1, 2, 3, 4, 5, 6, 7, 8, 9)

println("Average: " + ds.average) // 5.0 (double value)
println("Minimum value: " + ds.minValue) // 1 (int value)
println("Maximum value: " + ds.maxValue) // 9 (int value)
```

Listing 4.7: Scala statistics DSL basic example.

[24] `http://richhickey.github.com/clojure/clojure.inspector-api.html`

As a first example of using implicit conversion to apply a DSL to new values, we want to compute statistic values for an array of words (strings) using the data-set DSL. In the following example an implicit conversion from string-array to data-set is utilized to find the average, minimum and maximum word lengths. The data-set is created using the length of the strings as input, in the implicit function `str2DS`. It is then possible to call data-set methods on the string-array as if it was an instance of a data-set (which in fact it is as the compiler implicitly converts it to one on demand). The example is shown in listing 4.8.

```
val words = Array("eivind", "test", "oslo", "bekk", "scala", "elephant", "cat")

implicit def str2DS(str: Array[String]) = {
  DataSet(str.map(_.length): _*)
}

println("Average: " + words.average) // 4.86 is the average word length
println("Minimum value: " + words.minValue) // 3 is the length of the shortest word
println("Maximum value: " + words.maxValue) // 8 is the length of the longest word
```

Listing 4.8: Scala word statistics example.

In the next example we combine the image processing DSL with the statistics DSL using the same technique that was shown for the string-array above. An implicit conversion is defined converting instances of `GrayScaleImage` (image storing only one integer value per pixel) to a data-set, making it possible to call the statistics method to compute the average, minimum and maximum pixel values. In a way one could say that we have defined a new "statistical image processing DSL" combining the two existing DSL, all without writing any other code than the actual conversion. The whole example is shown in listing 4.9. The DSL creator would typically provide the function performing the implicit conversion (shown at the top), and the user would be able to call methods belonging to the statistical DSL directly on an image object (bottom part of code listing).

```
// Function must be within scope, typically as a result of a member import or package object
implicit def img2DS(img: GrayScaleImage) = {
  DataSet(img.data.toArray: _*)
}

// Example use:
val img = Image(Matrix(3, List(
  9, 8, 7,
  6, 5, 6,
  7, 8, 9)))

println("Average: " + img.average) // 7.22 is the average pixel value
println("Minimum value: " + img.minValue) // 5 is the smallest pixel value
println("Maximum value: " + img.maxValue) // 9 is the biggest pixel value
```

Listing 4.9: Scala example using implicit conversion to combine DSL for statistics with DSL for image processing.

As with the string-array example shown previously we see how easy it is to mix in the functionality of one DSL with another. One could imagine a whole environment built from small DSLs, each describing a specialized part of the domain. For example an environment for various numeric analysis could be built combining DSLs for image processing, statistics, charting etc. The image processing DSL can also be combined with the charting DSL, to visualize properties of images. The example in listing 4.10 on the facing page loads an image, and uses the charting DSL to draw an area diagram of its data. An implicit conversion creates the data-set needed for the charting DSL, counting pixels of the various values, thus resulting in a sort of histogram over the various gray values found in the picture. Figure 4.1 on page 53 shows two images together with their histograms, produced with the given code. As we can see, histograms are a useful visualization of the pixel distribution in an image. The low contrast of the first image is clearly shown with a narrow peak in the histogram, while the better contrast of the second image is shown with a more evenly distributed histogram.

4.2.2 Package Templates

Package templates is an object-oriented mechanism being research by the SWAT project at the Department of Informatics, at the University of Oslo. Package templates extend the notion of packages in languages like Java and

```
// Function provided by the DSL designer defining the conversion
implicit def img2DS(img: GrayScaleImage) = {
  val dataset = new DefaultCategoryDataset
  val imgData = img.data.toList
  for(i <- img.min to img.max) {
    dataset.setValue(imgData.count(_ == i), "img", i)
  }
  dataset
}

// Example use - load an image and use it directly as input to an area chart.
val img = loadImageCP("/numbers.png")

val histogram = createHistogram("Image Histogram", img)
```

Listing 4.10: Scala image histogram example

Scala, with the ability to instantiate and customize packages. The mechanism is described in detail with examples in [2]. This section outlines how a package template mechanism can be utilized to combine embedded DSLs.

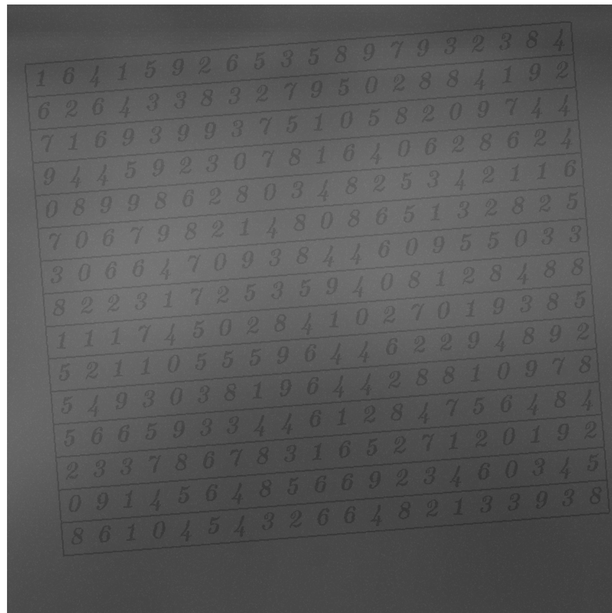
Having a package template mechanism in Java could be used as a sort of multiple inheritance, as the mechanism supports merging implementation details from several package classes into one new class. This could be used to combine DSL classes similar to the example shown with implicit conversions in the previous section. An example of how this could be done is shown in listing 4.11. Notice how the `ImageStatistics` class constructed has methods from both the `Image` and `DataSet` super classes, similar to what could be achieved using traits with Scala.

```
template Statistics {
  class DataSet {
    public int avg() { // Example statistical operation
      int[] data = getData();
      ... // Lots of Java code to compute and return average value
    }

    abstract int[] getData();
  }
}

template ImageProcessing {
  class Image {
    int[] imgData;

    public void blur() { ... } // Example image operation
  }
}

inst Statistics with DataSet => ImageStatistics
inst ImageProcessing with Image => ImageStatistics

class ImageStatistics with Operations {
  int[] getData() { return imgData; } // Return image data as an array
}

// Examples of use, an ImageStatistics object has both blur() and avg() methods:
ImageStatistics img = ...
img.blur();
int average = img.avg();
```

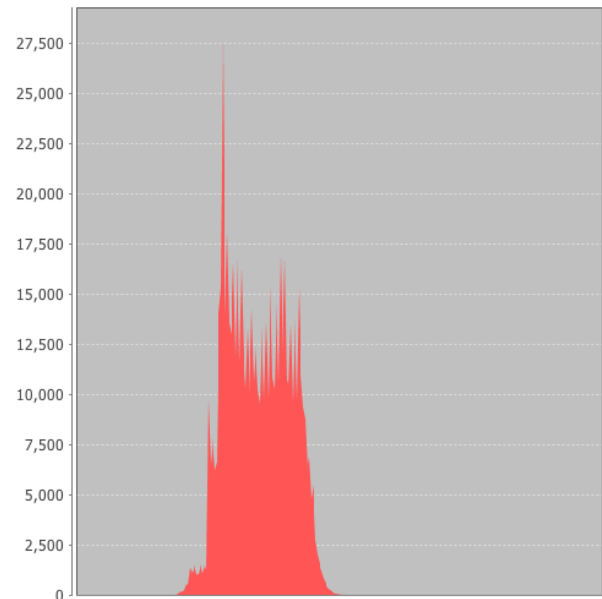
Listing 4.11: Package template DSL example.

4.3 Summary

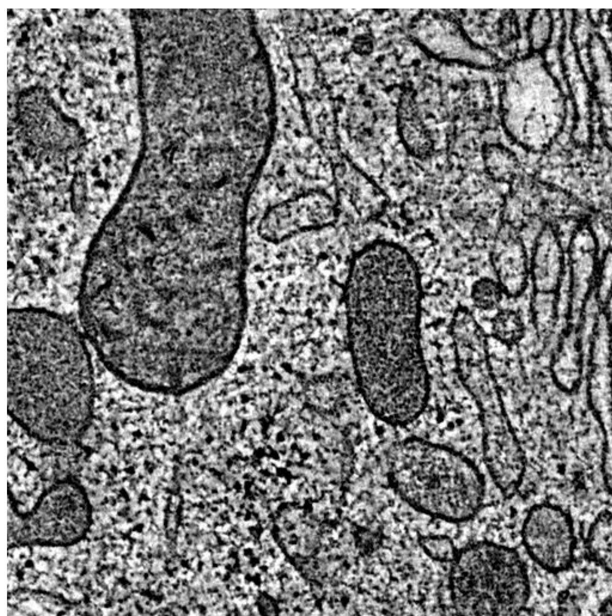
This chapter has explored ways to create customizable DSL components, as well as techniques used to combine several DSLs. Mixin-composition, found in Groovy, Ruby and Scala, is a way to create loosely coupled modular DSLs. New versions of the DSL can be created easily only varying on the specific traits included in the actual implementations. Another approach to creating generic DSLs is using type parameters or abstract types. The functional composition techniques in Clojure provide some exiting new ways to compose modular code, compared to the more imperative languages. Implicit conversion is a good technique found in Scala that enables the combining of two DSLs in a smooth manner. The package templates mechanism research at the University of Oslo also seems like a good alternative to combining DSLs.



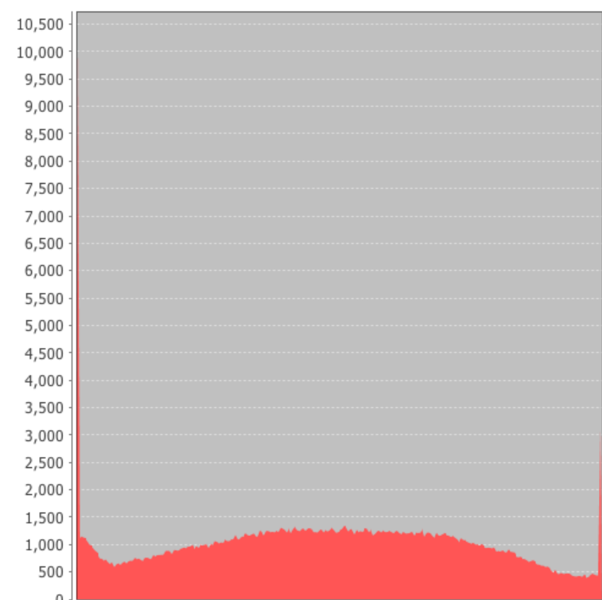
(a) Image I



(b) Histogram I



(c) Image II



(d) Histogram II

Figure 4.1: Two different images together with their generated histograms, showing the distribution of gray levels in the images. It clear from the histogram that Image II uses a much larger part of the pixel values available, which means it has a higher contrast, than Image I.

Chapter 5

Changing the Host Language

In the previous chapter we looked at customizable DSLs. This chapter deals with a related topic, namely customizing the host language that the DSL is implemented in. The chapter is divided in two main parts. The first section discusses ways to remove (or "prune") features from the host language, while the last section shows how the various host languages can be extended. This chapter does not rely on a big example, but shows through small stand-alone examples what can be done with the languages.

The reason for looking at this requirement is to see how much it is possible to customize the host languages. There are several situations where this is useful. Maybe the host language has some feature that is likely to cause errors if used together with the DSL. In that situation, it would be useful to disable or remove that feature in the scope of the DSL. This would "protect" the user from doing something wrong. At the same time it must be done with care, as users who are familiar with the host language may get offended or confused if features are missing.

Another situation is that where the DSL uses some kind of pattern that one would like to include in the host language. If features can be added to the host language, maybe the syntax feels more native, or maybe one could benefit from added compiler-checks or similar? Maybe extensions to the host language can be used in combination with pruning unwanted features, making the DSL appear almost as an external DSL with its own parser and interpreter?

5.1 Removing from the Host Language – Pruning

The following section describes how features can be removed from the different programming languages. Where possible, small examples have been implemented to show how the mechanisms can be used.

5.1.1 Compiler Plugins in Scala

Scala supports the notion of compiler plugins. This means that you can write your own code that will be run as part of the compilation process in Scala. The compiler plugin has full access to the abstract syntax tree, and can make changes or additions as needed. The plugin is packaged as a separate unit, and does not impact the main Scala distribution. So if the plugin is present the functionality will be available, but once it is removed the added functionality will not be available anymore.

A compiler plugin could be used to remove or disable features in Scala, although primarily used to add new features (extending the host language). For example if you want the DSL to be used in a purely functional programming setting, features of scala supporting imperative programming may be disabled. A compiler plugin can be made to return an error if constructs like `while-loops` (functional programming uses functions and recursion instead of pure loops) or `var` (functional programming uses immutable values defined by `val` instead of `var`) are used.

Listing 5.1 on the next page shows an example implementation of a compiler plugin checking for divide by zero

errors.[25] This is an example of host language extension, but a plugin removing functionality could be made in a similar manner. Compiling code with this plugin activated renders an error if division statements have the integer 0 as divisor (as line 5 of the file `Test.scala` does in this example):

```
\$ scalac -Xplugin:divbyzero.jar Test.scala
Test.scala:5: error: definitely division by zero
  val amount = five / 0
                    ^
one error found
```

```
package localhost

import scala.tools.nsc
import nsc.Global
import nsc.Phase
import nsc.plugins.Plugin
import nsc.plugins.PluginComponent

class DivByZero(val global: Global) extends Plugin {
  import global._

  val name = "divbyzero"
  val description = "checks for division by zero"
  val components = List[PluginComponent](Component)

  private object Component extends PluginComponent {
    val global: DivByZero.this.global.type = DivByZero.this.global
    val runsAfter = "refchecks"
    val phaseName = DivByZero.this.name
    def newPhase(_prev: Phase) = new DivByZeroPhase(_prev)

    class DivByZeroPhase(prev: Phase) extends StdPhase(prev) {
      override def name = DivByZero.this.name
      def apply(unit: CompilationUnit) {
        for ( tree @ Apply(Select(rcvr, nme.DIV), List(Literal(Constant(0)))) <- unit.body;
             if rcvr.tpe <:< definitions.IntClass.tpe)
          {
            unit.error(tree.pos, "definitely division by zero")
          }
      }
    }
  }
}
```

Listing 5.1: Scala “Divide-by-zero” compiler plugin.

Using compiler plugins has the advantage that you can have full control of what features to remove/add to the host language. The disadvantage is that you need to make sure the compiler plugins are used, either by specifying a separate compile command including the plugin jar files or depending on the users actually including the plugin when compiling. If the DSL is used in some kind if integrated development environment, it may be difficult to ensure that the plugin is actually included.

5.1.2 Pruning in Dynamic Languages

Another way to prune functionality from the host language is to utilize the open classes in Ruby, first introduced in section 3.13 on page 43. This can be used to redefine methods on the base classes in the language, maybe throwing an exception or printing an error message if an unwanted method is called. As an example, the code in listing 5.2 on the next page shows an example where the implementation of `String.length` is redefined to print an error message.

Ruby also has some metaprogramming methods that can be used to remove methods from base classes. The `remove_method` method can be used to remove a given method from the current class, while `undef_method` will

[25] Example taken from: “Writing Scala Compiler Plugins” – `http://www.scala-lang.org/node/140`

```
class String
  def length
    puts "Illegal method invoke!"
  end
end
```

Listing 5.2: Ruby String pruning example.

undefine the method completely. In listing 5.3 the `String.length` is undefined completely, rendering the message «NoMethodError: undefined method 'length' for "hei":String» if called on the string "hei".

```
class String
  undef_method: length
end
```

Listing 5.3: Ruby `undef_method` example.

Similar ways to modify the base API methods and classes are also available in Groovy, as was shown with the `random` method redefinition in listing 3.34 on page 41.

The advantage with using the approach offered in the dynamic languages is that the changes can be made in code, and included directly in the DSL. All functionality that is based on classes and methods can be changed. The limitations with the approach is that mechanisms built into the language are impossible to remove. Removing support for basic forms like `while-loops` or `if-else` statements can not be done.

5.1.3 Pruning in Functional Languages

Clojure has very few built-in control structures and keywords. Much of the core functionality is defined as macros or functions in the `clojure.core` namespace. Since Clojure is a dynamic language, all of these macros can be redefined by the user. The code in listing 5.4 shows an example where the core function for adding numbers is redefined to multiply instead. Although the example is pretty useless, it still shows how base language functionality can be changed. It is a powerful, but dangerous ability. Imagine the kinds of error users would get if this was done in a number-centric DSL like the image processing example used for this thesis.

```
; Enter the clojure core namespace
(ns clojure.core)

; Redefine the add method as multiply
(def + *)

; The result of this addition will surprise the user
(+ 2 3)
```

Listing 5.4: Clojure redefining core functions.

Clojure also supports removing mappings from namespaces completely. The code in listing 5.5 removes the mapping for the + function in the current namespace. Attempting to add numbers after this renders an error: «Unable to resolve symbol: + in this context». This is a mechanism that could be used directly as part of the DSL code. Users would still be able to redefine the things that have been removed, but the "default" would not to have them available.

```
; Remove the mapping of + for the current namespace
(ns-unmap *ns* '+)

; Generates an error
(+ 2 3)
```

Listing 5.5: Clojure undefining core functions.

Another way to change the core functionality of Clojure, is to use `dynamic bindings`. With dynamic bindings it is possible to redefine the meaning of a globally defined value, within the scope of a function/macro. An ex-

ample is given in listing 5.6[26], showing how the `noprint` macro redefines the `*out*` special variable to a dummy `java.io.Writer` object. The `*out*` variable is used by many functions to print information to a standard output/-console. However, because of the redefining done by the dynamic binding, all print statements within a `noprint` macro call will be suppressed. Using dynamic bindings is a nice way to prune or redefine features only within a limited scope.

```
(def bit-bucket-writer
  (proxy [java.io.Writer] []
    (write [buf] nil)
    (close []    nil)
    (flush []    nil)))

(defmacro noprint
  "Evaluates the given expressions with all printing to *out* silenced."
  [& forms]
  `(binding [*out* bit-bucket-writer]
     ~@forms))

(println "Regular println!")
(noprint (println "Noprint println!"))
```

Listing 5.6: Clojure macro binding core variable.

An advantage with the pruning possibilities found in Clojure is, as for the other dynamic languages, that the changes can be made directly in code and included in the DSL. It is the only language where base features like conditional logic or similar can be disabled directly in code (without using compiler plugins or other “external” mechanisms). A problem with Clojure may be that it allows too many changes. If not used carefully, one may end up removing base features that can cause other features to malfunction.

5.2 Extending the Host Language

This section discusses extensions to the host language. For some of the languages, the same features that were shown for pruning can also be used for extending the language. These are only mentioned briefly, with references to the previous section. Some new mechanisms are discussed, with a focus on how they aid in extending the host language.

5.2.1 Compiler Plugins

The compiler plugin mechanism described in 5.1.1 on page 55 may of course also be used to extend the language. Some of the new functionality added to Scala is first introduced as compiler plugins, and then later added to the main distribution after being tested. An example of this is the continuations support that will be added to Scala 2.8[27], and has been available as a downloadable plugin some time before the new version is released.

A similar approach to that of the Scala language itself (letting users try new functionality as downloadable plugins), could be done with a DSL. Features could be enabled or disabled based on the plugins included in the compiler.

5.2.2 Dynamic Languages

The mechanisms described in section 5.1.2 on page 56 can just as well be used to extend the host language with new methods. Existing classes can be modified, even final or core language classes. Examples of this were shown in section 3.13 on page 43.

5.2.3 Functional Languages

The functional languages have a few different ways to extend the host language. The first subsection below describes how functions themselves can be implemented to appear as built-in keywords. This is not really an ex-

[26] Example from `http://en.wikipedia.org/wiki/Clojure#Examples`.
[27] A Taste of 2.8: Continuations – `http://www.scala-lang.org/node/2096`

tension of the language, but still included here as it offers some of the same possibilities. The second subsection describes Clojure macros, a flexible mechanism when it comes to extending the language.

First-class Functions as Built-in Control Structures

With functional languages it is possible to create control structures that look like built-in keywords, but are simply first-class functions. In Scala functions can be defined with "pass by-name" parameter passing, and single parameter functions can be called using curly braces instead of parenthesis to specify the parameter (an example was shown in section 3.8 on page 37). As an example consider the definition of a new while construct `mywhile` shown in listing 5.7. The function `mywhile` is implemented as a curried function (having two lists of parameters, instead of one list with two parameters), using recursion to call itself as long as the given condition evaluates to true. The usage (shown in the bottom part of the listing) is exactly the same as the regular while construct. This is partly due to the fact that Scala allows both regular parenthesis and curly braces to surround function arguments. Notice how the condition is given in parenthesis, and the body statements in curly braces, although they are both just by-name parameters.

```
def mywhile(cond: => Boolean)(body: => Unit) {
  if(cond) {
    body
    mywhile(cond)(body)
  }
}

// Example use:
var i = 0
mywhile(i < 3) {
  println("Num: " + i)
  i = i + 1
}
```

Listing 5.7: Scala example making functions look like built-in keywords.

As we can see it is simple to define functions that act and look like built-in control structures. This is useful designing DSLs where we want it to seem like the host language has been extended with new features supporting the DSL. So this is not really a genuine extension of the language, but something that appears to be and does the same job in many cases.

Clojure Macros

In Clojure it is also possible to create functions as in the Scala example in the previous section. But Clojure has a much more powerful mechanism as well, namely the macro system. Macros in Clojure extend the compiler by writing regular user code, and are totally different from other programming languages macro systems. Many of the core constructs in Clojure are in fact implemented as macros, and are not actually primitives or other built-in support. Macros were introduced in section 3.12.3 on page 42, with a focus on metaprogramming. In this section they are used to show how Clojure, as a host language, can be extended.

A macro can be seen as a kind of template that outputs code, like that found in many other programming languages. The big difference with macros in Lisp-languages like Clojure however, is that the macro itself is valid code and has full access to the whole language. As a first example, consider the Clojure `unless` implementation as a macro shown in listing 5.8.

```
(defmacro unless [expr form]
  (list 'if expr nil form))
```

Listing 5.8: Clojure macro adding unless

To further exemplify the power Clojure macros bring when it comes to extending the host language, consider the macro in listing 5.9 on the following page. What it does is revert the order of the statements given as arguments. The `let` statement in the beginning defines a temporary value `rev#` as the reverse value of the list of arguments (`& body`). This reversed list is then splice-quoted back as code together with a Clojure `do` statement. An example

using the macro is shown in listing 5.10. As we can see the list of statements first provides a multiplication of two values (x and y), then the values are defined.

```
(defmacro rev [& body]
  (let [rev# (reverse body)]
    `(do ~@rev#)))
```

Listing 5.9: Clojure macro reverting statements order.

```
(def result
  (rev
    (* x y)
    (def y 4)
    (def x 3)))

(println (str "Result: " result))
; Result: 12
```

Listing 5.10: Clojure reverse statements example

The example with the reverse macro shown above may seem pretty useless, and it probably is. However, it was included here just to show some of the power made available by the Clojure macro support. It would probably prove difficult to write anything like that in any of the other languages studied in this report. Clojure gives you huge possibilities in extending/changing the host language. The uniform and simple syntax together with the macro system lets you build your own language, however you want it. Just about any control structure can be built straight into the language using macros.

5.3 Summary

There are different ways to modify the host language. With compiler plugins, as supported by Scala, it is possible to redefine whatever you like in the language. This does however require the use of externally activated plugins, that must be available whenever the functionality they provide is wanted. Dynamic languages support ways to add methods to existing base classes, as well as removing or redefining methods.

The most exiting capabilities in changing the host language is found using macros in Clojure. They are an integrated part of the language, but can still redefine existing functionality or add all kinds of new structures. Much of the base functionality in Clojure is actually built using macros, as opposed to being integrated as separate keywords.

Chapter 6

Exploring Ways to Use Embedded DSLs

So far the focus has been on the design/implementation of embedded DSLs. This chapter discusses various ways to use embedded DSLs. There are quite many ways to take advantage of a DSL, depending on what kind of language it has been implemented in. The next sections describe some of the usages identified, and how they are supported by the different languages.

6.1 API/Library

The most common way to use an embedded DSL seems to be as a library integrated in other application code. On the JVM this will typically involve archiving compiled class files, or script + interpreter, in one or more jar (Java Archive) files. These files will be included in the classpath of the application that needs access to the library API. The application programmer will then be able to import the classes and use in the program code. An example is the JMock framework in Java (also mentioned in section 3.1 on page 29). It allows programmers writing unit tests to specify with an internal DSL syntax how they want to mock dependencies to the code they are testing. An example implemented in Java (DSL syntax mostly found in the expectations declarations) is shown in listing 6.1.

```
import junit.framework.TestCase;

import org.jmock.Mockery;
import org.jmock.Expectations;

public class PublisherTest extends TestCase {

  public void testSimplePublish() {
    Mockery context = new Mockery();
    Subscriber subscriber = context.mock(Subscriber.class);
    Publisher publisher = new Publisher();
    publisher.add(subscriber)
    String message = "message";
    context.checking(new Expectations() {{
      oneOf(subscriber).receive(message);
      allowing(subscriber).isAlive(); will(returnValue(true);
    }});
    publisher.publish(message);
    context.assertIsSatisfied();
  }
}
```

Listing 6.1: Java example using jMock as an API.

Another interesting thing when using the DSL as a library, is that the language using the DSL not necessarily needs to be the same as the host language the DSL was implemented in. If the DSL is compiled to bytecode (`.class` files), it can be used from most of the other languages running on the JVM. The example above could just as well

have been implemented in any of the languages used in this thesis, for example in Scala as shown in listing 6.2 (looking quite similar to the original Java example above).

```
import junit.framework.TestCase

import org.jmock.{Mockery, Expectations}

class PublisherTest extends TestCase {

  def testSimplePublish {
    val context = new Mockery
    val subscriber = (context.mock(classOf[Subscriber])).asInstanceOf[Subscriber]
    val publisher = new Publisher
    publisher.add(subscriber)
    val message = "message"
    context.checking(new Expectations {
      oneOf(subscriber).receive(message)
      allowing(subscriber).isAlive; will(returnValue(true)
    })
    publisher.publish(message)
    context.assertIsSatisfied
  }
}
```

Listing 6.2: Scala example using jMock as an API.

6.2 Scripting

A different approach to the use-as-a-library approach shown above is to allow code implemented in the DSL to be run as a standalone script. That is without the need to wrap it in some class or module implemented in the host language. All the dynamic and interpreted languages support this kind of usage. Scala, although it is a static and compiled language, has a kind of interactive interpreter called REPL (Read-Evaluate-Print Loop) that typically can be used to test code on the fly. It can also be used to run Scala-code in a more script-like way. The following script can be run demonstrating this functionality, using the image processing DSL from chapter 2 on page 15 as example (loading an image and saving a blurred copy to a new file):

```
import simage.io.SImageIO._
import simage.structs._
import simage.structs.StrElType._

val img = loadImageFile("cell.jpg")
val se = StrEl(Square, 3)

val blurredImg = img.avg(se)
saveImage(blurredImg, "cell_blurred.jpg")
```

Listing 6.3: Scala scripting usage example.

The script can then be run from the command line using the scala interpreter (called with the `scala` command), assuming the script is in a file named `ImageScript.scala` and that the image processing DSL has it's classes packed in the jar `simage.jar`:

```
scala -classpath simage.jar ImageScript.scala
```

For users of the image processing DSL this means a great simplification in the way it is run. They do not have to learn how to make classes, objects and methods in Scala, as they would if they wanted to run the DSL as a regular library in a Scala file (requiring at a minimum creating an application object with a main method). As image analysis experts are not necessarily computer programmers this may be a good idea.

Similar examples can be given for other DSLs and using other host languages supporting running code as a script.

6.3 Interactive Console – Interpreter

Many of the programming languages studied have some sort of interactive interpreter (like the Scala REPL mentioned in the previous section). This can in many cases be configured to be used with a specific library pre-loaded, which means it is possible to create a custom version for working interactively with a DSL. Similar to the scripting example shown in the previous section this opens possibilities to work with the DSL without wrapping it in any host language code. But the main advantage of using the DSL in an interactive console is the ability it offers in issuing commands that are run directly.

The Scala REPL has an option to pre-load a file with commands on start-up. Typically this can be used to include imports, implicit definitions, type aliases or function definitions. Taking the example from the previous section on scripting we can put all the import statements into a file called PreDef.scala:

```
import simage.io.SImageIO._
import simage.structs._
import simage.structs.StrElType._
```

Listing 6.4: PreDef.scala file with image processing imports.

The interactive interpreter can then be started using the option `-i PreDef.scala` as in the following example:

```
scala -classpath simage.jar -i PreDef.scala

Loading Predef.scala...
import simage.io.SImageIO._
import simage.structs._
import simage.structs.StrElType._

Welcome to Scala version 2.7.7.final (Java HotSpot(TM) 64-Bit Server VM, Java 1.6.0_17).
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

Listing 6.5: Scala REPL execution with preloaded import statements.

We can then issue the same commands used in the script example one by one (notice the output for every line which is simply the interpreter calling the `toString` method after printing the type on the newly created object – useful for debugging purposes):

```
scala> val img = loadImageFile("cell.jpg")
img: simage.structs.GrayScaleImage with simage.operations.Standard with simage.operations.Morphology =
Image 512x512

scala> val se = StrEl(Square, 3)
se: simage.structs.StrEl[Int] =
Array2D rows:3 cols:3
Array(1, 1, 1, 1, 1, 1, 1, 1, 1)

scala> val blurredImg = img.avg(se)
blurredImg: simage.structs.GrayScaleImage with simage.operations.Standard with simage.operations.Morphology =
Image 512x512

scala> saveImage(blurredImg, "cell_blurred.jpg")

scala>
```

Listing 6.6: Scala REPL image processing example.

From Scala version 2.8[28] the REPL will have many improvements such as code completion (of package/class/member names) and interactive debugging.

[28] To be released in the first part of 2010, available as a first Beta version as of this writing.

As mentioned in the beginning of this section all the studied languages support interactive interpreters. Clojure has a REPL, much like that shown for Scala. Ruby has an `irb` (Interactive Ruby), and Groovy provides a Groovy-shell. These tools give all the languages some advantages over using Java directly. The ability to directly execute statements is typically useful when you need to test a piece of code. Most of the minor examples shown in this report where implemented directly in the interactive interpreter environments, and then copied out when working.

6.4 Embedded Interpreter

If we want to take the interactive interpreter in Scala a step further it is actually possible to embed it in your own application[29] [30] (see footnotes for some examples). This could be a way to create a standalone application having a console or similar being able to run DSL commands directly. For the image processing example shown in chapter 2 on page 15 it would be possible to create some kind of application combining such a console with an image viewer window showing the results of the operations directly on an image. This is similar to how MATLAB® works, a proven environment for image analysis and other mathematical calculations.

I think this technique could be useful for a number of cases. All kinds of DSLs where you may want to see some immediate results of your commands could be candidates. Examples could typically be statistical, financial or graphical DSLs. A generic application may even be reused for several different DSLs, thus being a base for building custom running environments for any embedded DSL implemented in Scala (or any other language that compiles to regular Java bytecode). A project using this approach is Kojo[31], a programming learning environment for children. They have their own Scala-based DSL for programming in a visual environment, utilizing an embedded REPL for writing and executing commands. Figure 6.1 on the next page shows a screenshot of Kojo, with the embedded REPL in the bottom left window.

6.5 Tool Support

Since working with an embedded DSL to a large extent means dealing with a host language, I think it is interesting to provide a brief overview and comparison over the development tool support the various languages offer.

6.5.1 IDE – Integrated Development Environments

An important productivity factor when programming is having access to a good IDE (Integrated Development Environment), helping with editing, running, refactoring and organizing the code. Java has come a long way in this area, with products such as Eclipse, NetBeans and IntelliJ IDEA being central contributors to developer effectivity. Plugins exist for all the three mentioned IDEs giving some support to all the languages used in this thesis. For example the Image Processing DSL was to a large extent implemented using the IntelliJ IDEA environment, with a Scala plugin (screenshot shown in figure 6.2 on page 66). A thorough comparison between the various alternative environments have not been carried out, but as a general comment it is interesting to see the difference between static and dynamic languages. Static languages seem to be easier to implement tools for, as much of the functionality is to some degree based on type evaluations (for example code completion, error notifications and refactoring).

6.5.2 Command Line Tools

The Java platform has a large amount of command line based tools available, both from Sun and third party providers. Many of these tools can be used more or less directly on all languages running on the JVM, as they can analyze bytecode or other JVM-based management options. Typical examples are standard Java tools like `jps`, `jstat`, `jstack` and `javap` (all included in standard Java Development Kit distribution.[32])

Other useful tools include those that are used to automate the building of projects (like `make`[33] for unix). All the

[29] http://suereth.blogspot.com/2009/04/embedding-scala-interpreter.html
[30] http://speaking-my-language.blogspot.com/2009/11/embedded-scala-interpreter.html
[31] http://www.kogics.net/sf:kojo
[32] http://java.sun.com/javase/6/docs/technotes/tools/
[33] http://www.gnu.org/software/make/

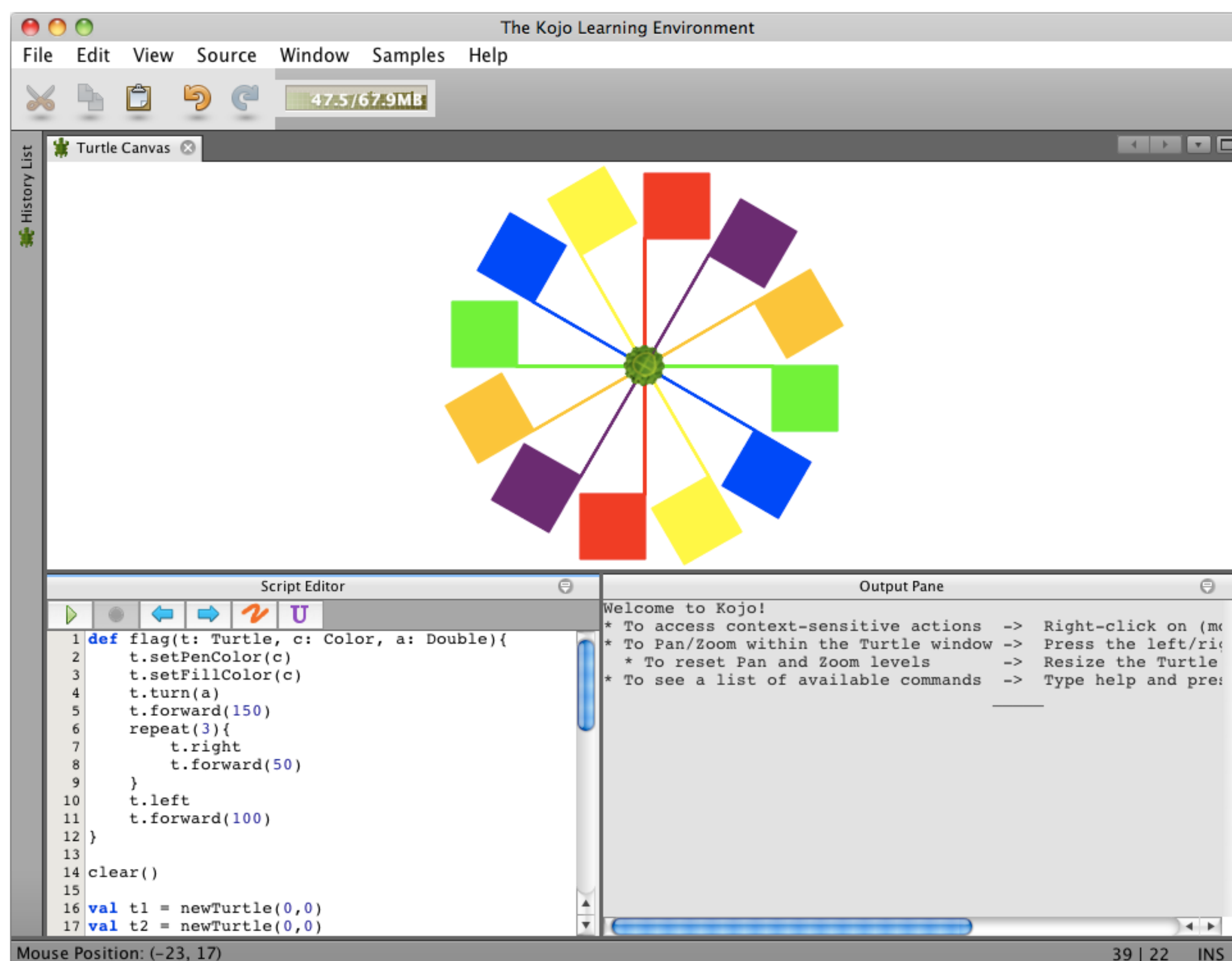


Figure 6.1: Screenshot from Kojo, a programming learning environment with an embedded DSL-REPL.

languages studied have respective alternatives, like `rake`[34] for JRuby, `sbt`[35] for Scala and `leiningen`[36] for Clojure. In addition many of the basic Java tools also work for alternative languages. The Image Processing DSL was compiled and tested using `Maven2`[37], a commonly used open source tool for building Java-projects.

6.6 Error Handling

This section discusses error handling possibilities in the different programming languages. When embedding a DSL it is important to control how exceptional situations are handled, and how errors are presented to the user of the DSL. The first sub-section shows how the different languages relate to the exception mechanism found in Java, and the following sections discuss differences related to language category.

6.6.1 Exception Handling

The Java programming language uses exceptions to handle errors and other exceptional events.[38] It is a basic mechanism where code can be wrapped in a `try` block, and any exception thrown can be handled in separate

[34] `http://rake.rubyforge.org/`
[35] `http://code.google.com/p/simple-build-tool/`
[36] `http://github.com/technomancy/leiningen`
[37] `http://maven.apache.org/`
[38] `http://java.sun.com/docs/books/tutorial/essential/exceptions/index.html`

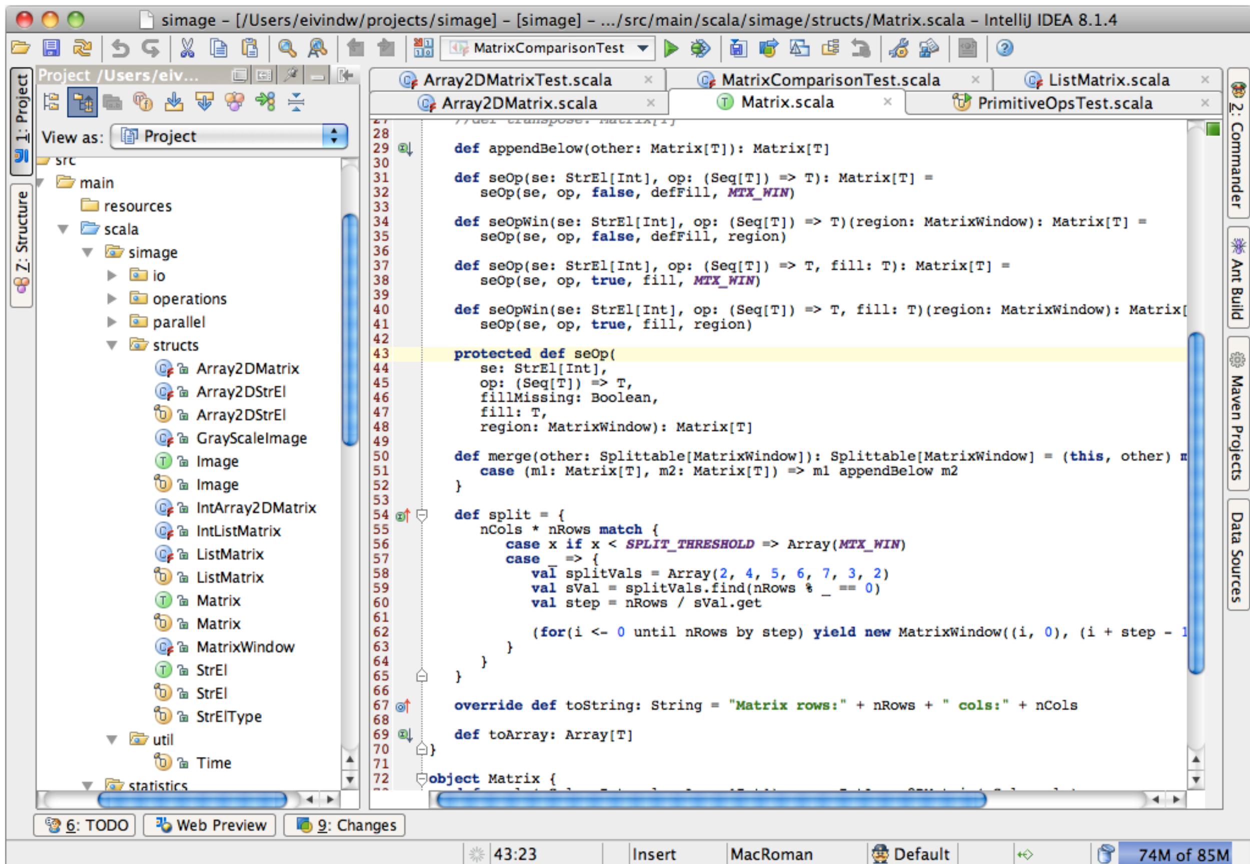


Figure 6.2: Screenshot from IntelliJ IDEA, an IDE having good support for Scala.

`catch` blocks (one for each type of exception). Additionally one can use a `finally` block for code that must be run no matter if an exception has been thrown or not. There are two main types of exceptions: checked exceptions that *must* be caught by the calling code (failing to do so will lead to a compile error in Java), and unchecked exceptions that *can* be caught by the calling code. This section will not go into details about the exception mechanism in Java, but will show how the different alternative programming languages relate to the exception mechanism.

In Scala, exceptions can be thrown in the same way as in Java [32]. Instead of returning a value, one can throw an exception. The catching of exceptions, however, is a little different. Scala uses its *pattern matching* functionality [37] to group several exception types together in the same `catch` block. The `finally` block is similar to that found in Java. Other differences in Scala exception handling is that all exceptions are treated as unchecked. Even checked exceptions from Java APIs will be converted to unchecked exceptions by Scala. A last point to note is that the try/catch/finally structure in Scala is able to yield a value, which is more "correct" according to functional programming as all code should yield values and use of mutable data-structures is discouraged. An example is shown in listing 6.7 on the next page (function creating a `URL` object from a `String` path, with default values returned on exception), demonstrating both the pattern matching and yielding of values.

JRuby has a similar approach to the checked/unchecked separation as Scala. All exceptions are treated like unchecked exceptions, and do not have to be caught. Exceptions that you want to handle can be done with the `begin-rescue` block in JRuby. An example is shown in listing 6.8 on the facing page, giving the exception-part of functionality similar to that shown for Scala in the previous example.

The Groovy syntax for handling exceptions is the same as that of Java. The only difference is that, like both JRuby and Scala, all exception catching is optional. They are all treated as unchecked exceptions that can, but do not have to, be caught.

```
def urlFor(path: String) =
  try {
    new URL(path)
  } catch {
    case mue: MalformedURLException => new URL("http://www.uio.no")
    case unknown => new URL("http://exceptionworld.com")
  }
```

Listing 6.7: Scala exception example.

```
begin
  url = URL.new(path)
rescue MalformedURLException => mue
  url = URL.new("http://www.uio.no")
end
```

Listing 6.8: JRuby exception example.

Clojure will also, like the three other languages, treat all exceptions as unchecked and optional [19]. There are two special forms, `try` and `throw`, that can be used to handle exceptions as in the other languages. The `try` special form includes `catch` and `finally` as argument clauses. An example is shown in listing 6.9, again implementing the same example as shown for Scala and JRuby.

```
(defn urlFor [path]
  (try
    (URL. path)
    (catch MalformedURLException _ (URL. "http://www.uio.no"))))
```

Listing 6.9: Clojure exception example.

Functional Alternatives

Even though the functional languages Scala and Clojure support normal exception handling, there are other alternatives that in many cases may be better. [32] describes the *loan pattern*, where a control-abstraction function opens a resource and “loans” it to a function before it closes the resource. The exception handling would then be done in this function providing a control-abstraction, and not by the calling function. An example is shown in listing 6.10, using the `withPrintWriter` abstraction to write a date to a file ("date.txt") without handling exceptions. This would be valuable in a DSL setting where we may want to hide “complicated” exception handling from the user.

```
// Control-abstraction function
def withPrintWriter(file: File, op: PrintWriter => Unit) {
  val writer = new PrintWriter(file)
  try {
    op(writer)
  } finally {
    writer.close()
  }
}

// Example of use
withPrintWriter(
  new File("date.txt"),
  writer => writer.println(new java.util.Date)
)
```

Listing 6.10: Scala control-abstraction example.

Similar to the example shown for Scala can be done with Clojure. There is a macro called `with-open` which provides a general control-abstraction for opening, using and safely closing resources. An example, similar to that shown for Scala, is shown in listing 6.11 on the next page.

```
(with-open [f (new java.io.FileWriter "date.txt")]
  (.write f (str (new java.util.Date))))
```

Listing 6.11: Clojure with-open example.

These examples show that there are good alternatives available in the functional world when it comes to exception handling. Providing users of an embedded DSL with good control-abstractions is an easy to understand way to hide exceptional situations.

6.6.2 Static or Dynamic Typing – Compiler- or Runtime Errors?

There is a significant difference between compiled and interpreted languages, when it comes to helping the user of the language in discovering errors in code. The compiled languages are run through a two-step process; first the compiler will check as much as possible while generating the bytecode without actually running the code, then the JVM will run the bytecode. Many basic errors, related to typing, syntax and similar, will be resolved by the compiler before the code is run. With an interpreted language all errors, even the basic ones, will go through to the actual running/interpreting of the code. This difference may be important for some kinds of DSLs.

The major difference between compiling or not is of course related to the typing discipline of the language. The statically typed languages are type-checked at compile-time, while the dynamically typed languages are type-checked at runtime. One aspect with this that is interesting is that many projects using dynamically typed languages seem to rely on automated unit testing to a much larger extent than similar projects using statically typed languages. This has been somewhat documented through studies of projects carried out by BEKK, in a master thesis by Hans-Christian Fjeldberg [10] and further in a small report written by me [36]. Projects based on Ruby on Rails have, in many cases, just as much code in automated tests as in production code. So one could say that the fantastic capabilities earned by using Rails (dynamic metaprogramming, method-missing and similar shown in chapter 3 on page 29) comes at the cost of having to write code to prevent errors the compiler usually would deal with in statically typed languages like Java and Scala.

6.7 Summary

DSLs on the JVM can be used in a variety of ways; as a library, script, directly in an interpreter or even embedded with the interpreter. All the languages studied, except Java, support some kind of interactive interpreter/REPL. This can be used to evaluate DSL code directly, without the need to set up a full environment with source files.

A major advantage running a DSL on the JVM is the large amount of tools available for the platform. Many of the tools available for the Java language can be used directly for other languages running bytecode on the JVM. The statically typed languages seem to have somewhat better support for Integrated Development Environments, than the dynamic ones.

Reporting errors, and exception handling is quite different in the languages studied, although the basic Java exception mechanism is used as a base for all of them. There is also a big difference between code that is compiled first, compared to code being run on an interpreter, when it comes to error handling. Static languages have an advantage, in that they can discover type errors earlier than is the case with dynamic languages. Dynamic languages seem to rely on automated unit tests to a larger extent, to cope with the disadvantage of not knowing the types up front.

Chapter 7

Performance

This chapter looks at the performance aspects of the various DSLs and host languages. The categories of languages are compared with regards to performance, and the image processing DSL example is analyzed with a focus on efficient implementation details. The final section of the chapter considers some key points when dealing with DSLs that need to run as fast as possible on the JVM.

7.1 Host Language – Dynamic vs. Static Typing

One of the language categorizations that potentially has a lot of impact on the performance on the JVM is whether the language is dynamically or statically typed. While static languages typically perform type checking at compile-time, dynamic languages will postpone most type checking until runtime. The JVM does, however, not support runtime type checking very well in the current version, although some work is being done to improve this. The next version of Java will most likely have much better support for dynamic languages[39], including a specific JVM instruction (`invokedynamic`) allowing optimizations of dynamic method invocations.

At the moment much of the dynamically typed languages on the JVM are implemented using reflection and synthetic Java types. This makes the code run significantly slower than that of the statically typed languages. Some work has been done to optimize this in compiled dynamic languages, or languages supporting some notion of just-in-time compilation. If the type is dynamic, but still obvious to the compiler, it is possible for the compiler to create bytecode that is using regular static types instead. However, a thorough knowledge of the languages and their compilers is needed to know exactly what code will be optimized and how it should be implemented.

Several existing benchmarks have been studied to verify that the above statements about dynamic languages being slower are actually true. It is a problem that there are very few official benchmarks available. The ones studied here are published on various Internet sites.[40] [41] [42] [43] Discussions seem to indicate that it is difficult to create objective benchmarks. JIT-compilation on the JVM makes the platform unreliable when running small standalone tests, and the techniques used to create examples vary. However, most tests show that dynamic languages are significantly slower than the static ones.

A mini-benchmark was created with the languages studied in this thesis, trying to show that dynamic languages are indeed slower than the static ones. The code is quite simple, just adding numbers in a loop of 10 million iterations. The implementations for the different languages are shown in figure 7.1 on the next page. The goal was to make the implementations as similar as possible. This succeeded for all languages except Clojure, which does not really have variables that can be updated, that I simply made do the same number of additions with the `time` macro. So it is really difficult to say if the Clojure measurement is comparable or not. The Clojure example was also optimized

[39]The Java Community Process is developing «JSR-292: Supporting Dynamically Typed Languages on the Java™Platform» (`http://jcp.org/aboutJava/communityprocess/edr/jsr292`)

[40]`http://shootout.alioth.debian.org/`

[41]`http://pegolon.wordpress.com/2008/01/24/quick-ruby-groovy-performance-comparison/`

[42]`http://www.mikeperham.com/2008/12/13/clojure-vs-ruby/`

[43]`http://www.fischerlaender.net/perl/poor-groovy-performance`

```
public class Add {
  public static void main(String[] args) {
    long before = System.currentTimeMillis();
    int i = 0;
    int j = 1;
    while(i < 10000000)
      i = i + j;
    long time = System.currentTimeMillis() - before;
    System.out.println("Java time: " + time + "ms");
  }
}
```

(a) Java

```
def before = System.currentTimeMillis()
def i = 0
def j = 1
while(i < 10000000)
  i = i + j
def time = System.currentTimeMillis() - before
System.out.println("Groovy time: " + time + "ms")
```

(b) Groovy

```
before = Time.new
i = 0
j = 1
while i < 10000000 do
  i = i + j
end
time =
  ((Time.new - before)*1000).round
puts("JRuby time: #{time.to_s}ms")
```

(c) JRuby

```
val before = System.currentTimeMillis
var i = 0
val j = 1
while(i < 10000000)
  i = i + j
val time =
  System.currentTimeMillis - before
println("Scala time: " + time + "ms")
```

(d) Scala

```
(def j 1)
(print "Clojure ")
(time
  (dotimes [i 10000000]
    (+ (int i) (int j))))
```

(e) Clojure

Figure 7.1: Mini benchmark – implementation in 5 languages.

with the `int` (type-hint) statements helping the compiler optimize the operation.

When running the code it seems that there are big differences between the dynamic and static languages. All code has been run directly from the command line, and without any kinds of optimizations. Running the code samples with a small bash-script gives the following output:

```
Java time: 8ms
Groovy time: 398ms
JRuby time: 1071ms
Scala time: 18ms
Clojure time: 186ms
```

The tests were run on a standard MacBook (2 GHz Intel Core 2 Duo processor, with 2 GB DDR3 memory). The languages studied in this thesis seem to come up in this order when compared, the version of the language used is shown in parenthesis:

1. Java (1.6.0) – original bytecode is the fastest.
2. Scala (2.7.7) – nearly as fast as Java.
3. Clojure (1.1.0) – uncertain about this one, but it seems a bit faster than the other dynamic languages.
4. Groovy (1.7.2) – significantly slower than Java, faster than JRuby.
5. JRuby (1.4.0) – the slowest of all the languages studied.

Both my own and other results clearly show that the dynamic languages are much slower on the JVM. This may change, as mentioned in the beginning of this section, with new features on the JVM and more optimized compilers.

7.2 Code Efficiency

When performing the same operation a large number of times, as is the case with many algorithms for Image Processing, there are various implementation related things that need to be considered. In this section I summarize the lessons learned with regards to tuning the code for maximum performance. As such most of this section will be targeted towards Scala.

7.2.1 Choice of data structures

The underlying data structure for images and Image Processing is the matrix. Since Scala does not include an efficient implementation of matrix I ended up testing a few different variants:

- **List of Lists** – A very logical implementation of matrix can be made by using Scala `Lists`. The idea for this implementation was inspired by a blog post by Jonathan Merritt.[44] A simple example of using a list of lists is shown in listing 7.1. However, this implementation proves to be slow in specific element access and iteration (see discussion below).
- **2D Array** – In Scala an `Array` will usually be faster than `List` because it is compiled to a native array in bytecode. It is also faster to implement as one flat array, rather than an array of arrays. This gives a much more efficient implementation with regards to element access and iteration. The implementation becomes slightly less straight-forward as we need to calculate the element positions in the array, as shown in listing 7.2.

Some basic comparison measurements were made, comparing a matrix based on an array vs. a matrix based on list. The output is shown below. On a 200x1000 element matrix the list based matrix was nearly 10 times slower than the equivalent array implementation when performing an anonymous structuring-element operation. When performing simple subtractions on the same structures the list implementation was a bit faster. However, it is quite clear that implementation of a matrix needs to be done using arrays to obtain anything close to acceptable performance for advanced operations.

```
Matrix rows:200 cols:1000
# Block "Array Matrix SeOp" completed, time taken: 778 ms (0.778 s)
# Block "List Matrix SeOp" completed, time taken: 6540 ms (6.54 s)
# Block "Array Matrix -" completed, time taken: 146 ms (0.146 s)
# Block "List Matrix -" completed, time taken: 98 ms (0.098 s)
```

```
class Matrix[T](val elements: List[List[T]]) {
  val nRows = elements.size
  val nCols = if(elements.isEmpty) 0
              else elements.head.size

  require(elements.forall(_.length == nCols))

  def apply(row: Int, col: Int): T = elements(row)(col)
}

val m = new Matrix(List(List(1, 2), List(3, 4)))
```

Listing 7.1: Matrix implementation based on List of Lists.

```
class Matrix[T](cols: Int, val elements: Array[T]) {
  require(elements.size % cols == 0)

  val nRows = elements.size / cols
  val nCols = cols

  def apply(row: Int, col: Int): T = elements(col + row * nCols)
}

val m = new Matrix(2, Array(1, 2, 3, 4))
```

Listing 7.2: Matrix implementation based on 2D Array.

[44] `http://jsmerritt.blogspot.com/2008/07/matrix-multiplication-in-scala.html`

7.2.2 Iterations – `for` comprehensions vs `while` loops

Many Image Processing operations require some kind of calculation performed for every element in the underlying matrix. As an example, consider the neighbour average operation (explained in section 2.2.1 on page 21) implemented using a general structuring-element operation show in listing 7.3. Having as efficient implementation of the `seOp` method as possible is critical for large matrices. So having some way of efficiently iterating over all elements of the matrix is essential. Scala provides two basic forms of iterating; `for comprehensions` and `while loops`:

- **for comprehensions** – In Scala, for comprehensions are implemented as a monadic combination of the methods `filter`, `map` and `flatMap`. This is a functional control structure allowing compact and readable code. An example of implementing a general structuring-element operation based on for comprehensions is shown in listing 7.4 on the facing page.
- **while loops** – For regular loops Scala provides the `while` keyword. Using this instead of the for comprehensions gives a much more iterative implementation, as shown in listing 7.5 on page 74.

A small test was constructed comparing three different implementations. The two implementations shown in the bullets above are compared, as well as a special version of the for-comprehension without yielding values. This version without yielding values still uses the for-comprehension to build the loop structure, but depends on imperative code with variables (like that of the while-loop) to keep track of the results. So in a way one could say that the three examples are showing steps in doing more or less functional style programming. The pure for-comprehension with yielding values is purely functional, with all values being constants. The while-loop is purely imperative, and the last one is something in between. The results below show clearly the performance cost of using for-comprehensions. Dropping the yield statements improves performance quite alot, but the while-loop implementation is by far the most efficient. This example was done on a 1000x1000 matrix. Increasing the matrix size also increased the differences in efficiency. From these results it is safe to conclude that the nice code you can write with the functional programming features of Scala may come at the price of efficiency.

```
1000x1000 matrix
# Block "While loops" completed, time taken: 1865 ms (1.865 s)
# Block "For comprehensions w/o yield" completed, time taken: 3507 ms (3.507 s)
# Block "For comprehensions" completed, time taken: 8090 ms (8.09 s)
```

```scala
class Matrix ... {
  def seOp(se: Matrix, op: (Seq[Int]) => Int) = { ... }
}

val matrix = ... // Create matrix - load image or similar
val se = new Matrix(3, Array.make(9, 1))
val avgMatrix = matrix.seOp(se, (seq) => seq.reduceLeft(_ + _) / seq.size)
```

Listing 7.3: Implementing matrix neighbour average using a general structuring-element operation.

7.2.3 Using JVM Primitives

In Java, there are a number of primitive types (byte, short, int, long, float, double, char and boolean). These are not objects, but special types with different precision. All of the primitive types have associated wrapper types that are classes, and represented as objects in the JVM. These wrapper classes are typically used whenever a primitive needs to be treated as an object (for example as keys in maps, or other collection classes). From Java 5 there is even support for «autoboxing», an implicit conversion between primitives and their wrapper types.

However, there are performance issues related to the wrapper classes. Using primitives is much faster than using wrappers: *It is not appropriate to use autoboxing and unboxing for scientific computing, or other performance-sensitive numerical code. An Integer is not a substitute for an int; autoboxing and unboxing blur the distinction between primitive types and reference types, but they do not eliminate it.*[45] As stated in the quote, it is critical to

[45] `http://java.sun.com/j2se/1.5.0/docs/guide/language/autoboxing.html`

```
def seOp(se: Matrix, op: (Seq[Int]) => Int) = {
  val w = se.nCols / 2
  val h = se.nRows / 2

  def seValues(row: Int, col: Int) = {
    for {
      x <- -w to w
      cx = col + x
      y <- -h to h
      ry = row + y
      if(cx >= 0 && cx < nCols && ry >= 0 && ry < nRows)
    } yield {
      elements(cx + ry * nCols)
    }
  }
  val range = for(i <- 0 until nRows; j <- 0 until nCols) yield {
    op(seValues(i, j))
  }
  new Matrix(nCols, range.toArray)
}
```

Listing 7.4: Implementation of a structuring element operation on a matrix using `for comprehensions`.

ensure that primitives are used for numerical code, like the image processing DSL in this thesis.

With languages like Scala and Clojure, there are no primitive types in the syntax. Every value is an object (even in Clojure, being implemented on the JVM). The Scala compiler will try to optimize where possible, as stated in [32]: *The Scala compiler uses Java arrays, primitive types, and native arithmetic where possible in the compiled code.* However, it is not always obvious when this optimization will be performed. Typically will generic collections have some problems, even though the type parameter used is a primitive like `Int`. Scala 2.8 will introduce some more features to help write code that ensures the use of primitives at the bytecode level. Most notably is the `@specialized` annotation, making the compiler produce specialized versions of generic classes [5]. As an example consider the following code, defining a class `MyList` with type parameter `T`. The compiler will produce two specific implementations of the class, one for `Int` and one for `Double`:

```
class MyList[@specialized(Int, Double) T] ...
```

Clojure also has some support for specifying primitive types, despite the dynamic typing used throughout the language. This is called giving the compiler "type hints" [19], and can be used for optimization. A simple example is shown adding two integers, first the regular Clojure way:

```
(+ 2 3)
```

The code line above would be using reflection or other similar mechanisms used in all the dynamic languages. However, Clojure can be used with type hints that will make the code much more efficient:

```
(+ (int 2) (int 3))
```

With code that requires maximum performance on numerical calculations it is important to be aware of these mechanisms. A good thing is that one can always write code without thinking about performance first, and then add specializations, type hints or other mechanisms whenever you see a need to improve the performance. Another approach that could be done is to write the performance-critical code parts in regular Java code, and simply include the compiled Java-classes in the classpath of the programming language in use.

7.3 Concurrency – Parallel or Distributed Computing

In the following sections we look at different mechanisms available from the Scala language to achieve better performance through utilization of the hardware resources available in the computer. One of the claimed benefits using functional programming languages is that concurrent programming is made easier and more available. After a

```
def seOp(se: Matrix, op: (Seq[Int]) => Int) = {
  val w = se.nCols / 2
  val h = se.nRows / 2

  val newArr = Array.make(arr.size, 0)
  var points: List[Int] = Nil
  var j, i, x, y, cx, ry = 0
  while(j < nCols) {
    i = 0
    while(i < nRows) {
      points = Nil
      x = -w
      while(x <= w) {
        cx = i + x
        y = -h
        while(y <= h) {
          ry = j + y
          if(cx >= 0 && cx < nCols && ry >= 0 && ry < nRows) {
            points = elements(cx + ry * nCols) :: points
          }
          y = y + 1
        }
        x = x + 1
      }
      newArr(i + j * nCols) = op(points)
      i = i + 1
    }
    j = j + 1
  }
  new Matrix(nCols, newArr)
}
```

Listing 7.5: Implementation of a structuring element operation on a matrix using `while loops`.

quick discussion on the Java mechanisms to support concurrent programming the different functional paradigms are explored.

7.3.1 Java Concurrency

Java supports various abstractions that can be used when working with concurrency, all dealing with threads in some way or another.[46] Several threads can exist within a process, sharing resources like memory and files. This opens up possibilities for more efficient programming, but also opens up for errors regarding access to shared resources. With multi-threaded programming there are many problems that can occur, like deadlocks/livelocks (two threads blocking each other forever), starvation (thread waits for resource that is never made available) or data inconsistencies (several threads change shared memory without synchronizing with each other).

The main mechanisms provided in Java can be summarized in the following list:

- **Synchronization** – This is the lowest level of abstraction. Access to resources is synchronized so that only one thread can access it.
- **Immutable Objects** – This is more a programming style. If all programming is done in an immutable style, there will be no need to synchronize access to them as they can not be changed anyway.
- **Lock Objects** – More specific locking possibilities than using synchronization directly.
- **Executors** – Allocate pools of threads which are assigned tasks. This is a management facility, as synchronization between threads still needs to be taken care of.
- **Concurrent Collections** – Special collections that support atomic operations to add, change or remove values.
- **Atomic Variables** – Special objects that guarantee that changes will be performed in atomic operations.

[46] `http://java.sun.com/docs/books/tutorial/essential/concurrency/`

The next sections will explore how Scala and Clojure, the two functional programming languages, expand the possibilities on the JVM when it comes to concurrent programming.

7.3.2 Actors API

The primary concurrency construct in Scala is actors. Actors are basically concurrent processes that communicate by exchanging messages [18]. My first attempt at making the Image Processing DSL parallel was by making a generic executor using the Actors API. The result, with an example of simple usage, is shown in listing 7.6. Isolated parts of the processing can be split into separate functions that are distributed over a set of executors. So my example with a String operation is easily extended to Image Processing or any other domain that can be split into several parallel functions.

The main issue with this approach is that we need to wait for the result to be ready. This is of course not a big problem, but the next section describes another technique that is better suited for parallel computing.

```
import actors.Actor
import actors.Actor._

// Message contains data and transformation function
case class ExecMsg[T <: Any](data: T, op: T => T)

// Executor executes function and exits
class Executor extends Actor {
  def hasResult = result != None

  def getResult = result

  private var result: Option[Any] = None

  def act {
    while(true) {
      receive {
        case ExecMsg(d, op) => result = Some(op(d)); exit
      }
    }
  }
}

// Example usage - convert "scala" to upper case in separate thread
val exec = new Executor
exec.start // Start running - ready to receive messages
exec ! ExecMsg("scala", (s: String) => s.toUpperCase) // Send data and function
while(!exec.hasResult) {} // Busy waiting..
println("Result: " + exec.getResult.get) // Prints "SCALA"
```

Listing 7.6: A generic `Executor` based on the Scala Actors API.

7.3.3 Futures API

Actors may communicate using futures where requests are handled asynchronously, but return a representation (the future) that allows to await the reply. This is better suited for the parallel processing we are trying to achieve. An example similar to the pure actor example in the previous section using futures is shown in listing 7.7 on the next page.

Futures also support waiting for an array of functions to complete. This is the mechanism used in the Image Processing DSL. See listing 7.8 on page 77 for a complete example of how futures are used to run several operation in parallel and merge the results once all futures have finished running. The `Splittable` identifier is simply a trait specifying `split` and `merge` methods that need to be implemented by users of the `parallel` function. Notice how the futures are yielded to a sequence directly from the for comprehension and awaited completion using the `awaitAll` function.

```
import actors.Futures._

// Define function - could also be done directly in the future-block below
val func = (s: String) => s.toUpperCase

// Create future running the function with the String argument
val fut = future { func("scala" }

// Wait for and return the result
println("Result: " + fut())
```

Listing 7.7: An example using `Futures` to execute a function in a separate thread.

The general parallelization mechanism based on futures was used to implement concurrent support in the image processing DSL. The full code is too much to include here, but is available on GitHub. The `Matrix` trait[47] was made splittable, and could then be used to implement parallel operations on images. The code in listing 7.9 shows both a parallel and a non-parallel version of the average operation discussed several places earlier in this report. Worth noting is the small code difference needed to make it parallel. In a way one could say that the parallel mechanism itself is a small DSL for enabling parallelization in Scala. The next version of Java will most likely include a similar mechanism to that developed here with Scala, namely the Fork/Join framework described in [26].

```
trait Standard { this: GrayScaleImage =>
  def avg(se: StrEl[Int]) = {
    doParallelSeOp(data.seOpWin(se, (seq: Seq[Int]) => seq.reduceLeft(_ + _) / seq.size) _)
  }

  def avgSimple(se: StrEl[Int]) = {
    Image(data.seOp(se, (seq) => seq.reduceLeft(_ + _) / seq.size))
  }
}
```

Listing 7.9: Scala parallel image processing operations.

7.3.4 Functional Concurrency with Clojure

An important factor in all functional programming is the strict separation between mutable and immutable data structures. Where Scala, being a multi-paradigm language, offers mutable variables and encapsulation of state as all object-oriented languages (in addition to offering immutable alternatives), Clojure tries to be more pure on the functional side and hence makes immutable programming the "default" model in the language. All handling of state must be done in an explicit manner, forcing the user of the language to deal with all changes to the mutable state of a program. A program that has no shared mutable state can easily take advantage of concurrent programming. If you know that the same state can not be overridden by multiple threads there is no need for explicit locking or other mechanisms synchronizing access to the data. This is one of the major benefits when working with functional programming languages. This section shows how mechanisms for concurrency can be utilized in Clojure, with some simple examples.

One of the ways Clojure integrates with Java is in making all functions implement the `java.lang.Runnable` and `java.util.concurrent.Callable` interfaces, from the concurrent classes added to the JDK from version 1.5. This allows all Clojure functions being able to run directly in separate Java threads, or using executors or other mechanisms from the Java concurrent API. So spawning several threads to perform separate tasks is not difficult. An example is shown in the following listing, where the function printing "hello!" is run in a separate thread (also created directly) on the JVM:

```
(.start (Thread. (fn [] (print "hello!"))))
```

It gets a bit more complicated when several threads need to cooperate to solve a task, as in the Scala image processing examples shown in previous sections. Clojure provides some different mechanisms when working with

[47] `http://github.com/eivindw/simage/blob/master/src/main/scala/simage/structs/Matrix.scala`

```
import actors.Futures._

object Operations {

  def parallel[T](obj: Splittable[T], op: (T) => Splittable[T]): Splittable[T] = {
    obj.split match {
      case Array(region) => op(region)
      case regions: Array[T] => {
        val futures = for(region <- regions) yield future {
          op(region)
        }
        val results = awaitAll(5000, futures: _*)
        val parts = for(result <- results) yield result.get.asInstanceOf[Splittable[T]]
        parts.reduceLeft(_ merge _)
      }
    }
  }
}

// Example usage - double values in the list in separate threads
case class MyList(val list: List[Int]) extends Splittable[Int] {
  def split = list.toArray

  def merge(other: Splittable[Int]): Splittable[Int] = (this, other) match {
    case (t: MyList, o: MyList) => new MyList(t.list ::: o.list)
  }
}

val splitList = MyList(List(2, 5, 7, 1))

val doubledList = parallel (
  splitList,
  (i: Int) => {
    Thread.sleep(100) // Slow things down a little
    new MyList(List(i * 2))
  }
).asInstanceOf[MyList]
```

Listing 7.8: A general parallel mechanism based on futures.

shared data:

- **Atoms** – Single unit data that can be read/changed in atomic, synchronous operations.
- **Refs** – Shared data units that must be modified in Software Transactional Memory (STM), a Clojure transaction mechanism. Several refs can be changed in one atomic synchronous operation.
- **Agents** – Shared data units supporting asynchronous updates running functions on separate threads, using messages. This is similar to the Scala actor mechanism described in section 7.3.2 on page 75.

All of these mechanisms make it possible to program in a functional way, and apply the proper concurrency terms when needing to share state with other functions.

Another way to make Clojure code run faster is to use memoization of functions. With memoization the result of the function is cached in memory with the parameters as keys. Whenever a function is called with the same parameters as before the cached result is returned directly. So in using memoization on computationally heavy functions we are trading CPU for memory. Less CPU will be used, as the calculations only run once, but more memory will be consumed in caching all the function results. A simple example, inspired from an example in [19], of memoization is shown in code listing 7.10 on the next page. Running the example clearly shows that the memoized version runs roughly 3 times faster as it is actually only run two times for the given values:

```
\$ clj memo.clj
(slow-double) "Elapsed time: 602.931 msecs"
(mem-double) "Elapsed time: 200.744 msecs"
```

```
; Slow function
(defn slow-double [n]
  (Thread/sleep 100)
  (* n 2))
; Memoized version of slow function
(def mem-double (memoize slow-double))

; Timing the two functions over a vector of values
(def values [1 2 1 2 1 2])
(time (dorun (map slow-double values)))
(time (dorun (map mem-double values)))
```

Listing 7.10: Clojure memoization example

The memoization mechanism in Clojure would be useful when implementing an image processing DSL or similar requiring heavy computations. Large parts of many images are similar, and would most likely lead to the functions being called with the same parameters several times when passing over the image. A memoized version would simply return the already calculated value the second time around.

7.4 Secondary Processors – GPU

Recent years manufacturers of graphics cards and GPUs (Graphics Processing Unit) have made libraries available to run general code (not just graphics processing) on their hardware. This is typically hardware optimized for matrices and image processing, so it should be interesting to utilize this possibility to speed up the operations in a Image Processing library.

OpenCL™[48] is an open standard for parallel programming that utilizes the power of the GPU to perform calculations. OpenCL is being created by the Khronos Group with the participation of many industry-leading companies and institutions. A Scala API exists which enables Scala programs to make use of the possibilities offered by OpenCL – ScalaCL.[49]

A simple example using ScalaCL to add two arrays of integers is shown in listing 7.11 on the facing page. The `IntsVar` type is a special multi-dimensional data-structure storing integers (similar structures are available for floats and other data types), and the `:=` operator provides special operations that can be performed on the data-structures. In the example two 5-dimensional arrays are created and added together.

The ScalaCL library is still new and limited in functionality, but works for small examples. It is a new and exiting domain that will be important to keep up with in the future. Programs leveraging the power of the GPU will have vastly more computing power available than those who do not.

7.5 JVM Tuning – JVM Arguments

The JVM supports a wide variety of tuning/configuration options that must be considered when optimizing an application for performance. This section briefly describes some of these options, with a focus on the various programming languages running on the JVM. Configuring the JVM for performance is covered in details by documentation provided by Sun.[50] [51]

7.5.1 Configuring Memory Usage and the Garbage Collectors

The way memory is used and reused can be configured in a wide variety of ways on the JVM. This is common for all kinds of programs running on the virtual machine, regardless of language implementation. It is possible to control both the heap (where objects are stored) and stack (where active executions are stored) sizes, setting initial values and controlling how fast they can grow in size and the upper limits. Since the JVM provides automatic garbage

[48] http://www.khronos.org/opencl/
[49] http://code.google.com/p/nativelibs4java/wiki/OpenCL
[50] http://java.sun.com/javase/technologies/performance.jsp
[51] http://java.sun.com/performance/reference/whitepapers/tuning.html

```
import scalacl.{Dim, Program}
import scalacl.ScalaCL._

class SeOpOCL(i: Dim) extends Program(i) {
  val iarr = IntsVar
  val iarr2 = IntsVar

  var output = IntsVar

  content = output := iarr + iarr2
}

val arr1 = Array(1, 2, 3, 4, 5)
val arr2 = Array(5, 4, 3, 2, 1)

val prog = new SeOpOCL(new Dim(arr1.size))

prog.iarr.write(arr1)
prog.iarr2.write(arr2)

prog !

println("First: " + prog.output.get(0)) // Print 6
```

Listing 7.11: Add two arrays of integers using ScalaCL.

collection, setting memory limits directly control how often the garbage collector will run. Parameters are also available to control how the garbage collectors run.[52]

7.6 Summary

The performance part is one of the areas where there is a big difference between the various language categories. Dynamic languages are significantly slower than static ones running on the JVM, due to synthetic types and use of reflection-based mechanisms. This may change in future releases of Java as more native support for dynamic invocation is added to the JVM.

Functional languages provide an exciting new view on concurrency, with a focus on immutable data structures. Scala combines object-orientation with functional programming style to enable concurrency through a variety of mechanisms, such as actors and futures. Clojure provides a more pure functional view, enabling mechanisms like concurrent agents and function memoization to improve performance.

New frameworks are appearing allowing the use of GPUs or other secondary processors found on modern computers, potentially bringing the performance of JVM based programs to new heights.

Several ways to tune the JVM for performance exist. These could be used for most languages running on the JVM, although a more detailed understanding of each language should be made to utilize this in a best possible way.

[52] http://java.sun.com/docs/hotspot/gc5.0/gc_tuning_5.html

Chapter 8

Summary

This final chapter first provides a conclusion of the work performed and the results found. It ends with a section suggesting future work based on some of the results found.

8.1 Conclusion

This section concludes the results found during the work of the thesis. It is divided into subsection grouping related results on the same structure as the main chapters in the report. At the end follows a small section highlighting the key findings of this thesis work, with an overview of what kind of DSL tasks the different programming languages are suited for.

8.1.1 Language Categories and Embedded DSLs

There is no doubt that all the languages studied in this thesis provide many features that are not found in the Java language. Features that enable new and exciting capabilities regarding embedded DSL development on the JVM platform.

The dynamic object-oriented languages (Groovy and Ruby) offer a very concise coding-style, a central requirement when developing embedded DSLs. Code looks clean without information about types included, something that is valuable for many types of DSLs. A mathematician would maybe not care if the numbers are stored as doubles or ints, as long as the results are correct and as expected. Another mechanism only available in the dynamic languages is the dynamic metaprogramming described in section 3.12.2 on page 41. Code that dynamically generates new code, combined with dynamic dispatch in the form of the `method_missing` feature and monkey patching, provides a basis for popular DSL environments like the Ruby on Rails framework.

The multi-paradigm language Scala, combining object-orientation with functional programming, offers a huge set of new capabilities when it comes to embedded DSL design (compared to the Java language). Features like implicit conversions, type inference and operator notation on regular methods are just some examples. Being statically typed and compiled still makes the language similar in many ways to Java, and may be one of the easiest languages to learn for a Java programmer. Scala offers, in most cases, code that is just as concise as that of the dynamic languages, with the added benefit of being compiled and thus checked for many errors earlier than the interpreted languages.

The purely functional language of Clojure is maybe the most "different" from the rest of the languages, especially with regards to the syntax. With the inheritance from Lisp it has very few keywords and built-in control structures. However, it sports a macro-mechanism that is capable of creating most control structures. As such the language is maybe one of the best suited for writing narrow and specific DSLs, as it can be adjusted to exactly the way you want it. The language was included in the project as an "outsider", but proved its worth quickly.

8.1.2 Customizing DSLs

Mixin-composition is a mechanism enabling code reuse and modular design in a more precise way than in Java which only supports polymorphism through interfaces. All the object-oriented languages (Ruby, Groovy and Scala) support some form of mixin-composition. Most thoroughly studied was the traits in Scala, which were used to build the DSL for image processing. With the help of traits is was possible to create a modular DSL, that can be tailored to specific users needs.

Clojure supports multimethods. A mechanism enabling a much more detailed dynamic dispatch than that found in polymorphism. Multimethods can be used outside the object structure, giving the types new structure without altering them at all.

When it comes to combining DSLs the implicit conversion mechanism in Scala proved concise and easy to use. With only a few lines of code, converting from one data-structure to another, we are able to combine several different separate DSLs into one environment.

8.1.3 Modifying the Host Language

The languages studied offer a wide range of ways to modify the behaviour of their core functionality. For a DSL this is interesting, either to remove (prune) unwanted features or to add new control structures or other basic functionality.

In Scala, compiler plugins can be implemented to provide new or remove existing functionality. In addition, “pass by-name” parameters combined with functional style programming and member imports is a good way to write functions appearing as new control structures.

The dynamic languages support modifications of classes at runtime. This includes both removing methods from, and adding methods to, existing classes, even core Java classes marked as `final`.

The Clojure macro system is the most flexible way to build new control structures. Most of the core Clojure system is made up of macros implemented in Clojure itself, and can be redefined by the user if needed. With the macro system, it should be possible to create an embedded DSL appearing to be a separate programming language (like an external DSL).

8.1.4 Usage

While a DSL implemented in Java will have to be compiled and used like an API or library in other Java code, all the languages studied have other capabilities with regards to usage.

Interpreted languages can be run directly as scripts on a JVM, and the compiled ones include REPL-environments giving them the same capabilities. Creating and running a script-file is significantly faster than wrapping the functionality up in a class with a main method, as you would have to do in Java.

The interpreters/REPL-environments can also be used directly, to run code on the fly. This is an amazing capability when it comes to testing code out during implementation. Another exciting use is the ability to embed these environments directly in an application, providing an interactive environment for the DSL directly. This would be highly relevant for a numerical computation environment including the example DSLs of image processing, statistics and charting.

An advantage all the languages share is the ability to utilize the existing tools and frameworks on the JVM. Java is a popular language, and there are thousands of useful open source or other easily available tools to choose from. An example is the charting DSL that was implemented in Scala, but almost entirely based on an existing Java open source framework (JFreeChart).

When it comes to error handling all the languages support the exception mechanism being used in Java. In addition the functional languages support other ways to create control-abstractions, making the user less dependent on handling exceptions directly.

8.1.5 Performance

The biggest difference in performance between the language types is most likely that between dynamic and static languages. Dynamic languages running on the JVM are significantly slower, largely because of the use of synthetic types and reflection. This is due to the fact that the JVM was not created with dynamic languages in mind, but may change in future versions of the JVM, as the `invokedynamic` instruction is added to the JVM. But at the moment a DSL that needs effective code, like those adding thousands of numbers or similar (as the image processing examples designed in this thesis), should probably be implemented in a static language (like Java or Scala).

The functional languages provide ways to create very concise code. However, some of these features seem to cost quite a lot when it comes to performance. Some examples include for-comprehensions and pattern matching in Scala, that are much slower than while-loops and general if-else statements. So there is a trade-off between concise code and high performance. Scala 2.8 may improve this a little, with better support for JVM primitives and annotations to create effective recursions and pattern matching statements.

The functional languages of Scala and Clojure offer good mechanisms for concurrency abstractions. The immutable world of functional programming makes it easy to create threads that cooperate in solving problems. The biggest advantage is that they make concurrency easy to implement without risks for deadlocks and other common problems when working with Java concurrency.

A final note on performance is the support for GPU programming that is becoming available, even on the JVM. An example was shown where Scala was used to add arrays of integers using the power of the GPU instead of the main processor. This is something that potentially can make a lot of software run faster, as most modern computers are equipped with advanced GPUs capable of high numbers of numeric calculations.

8.1.6 Final Results – Programming Languages and Tasks

The goal in this thesis was to compare the various programming languages and find out what kind of tasks they were suited for. This section summarizes the key advantages of each language, and suggests an answer to when each language should be chosen.

I have tried to identify the points where the languages differ. The main advantages of each language, compared to the others, are summarized in the following list:

- **Java** – Java is the fastest language on the JVM, producing the most efficient bytecode.
- **Scala** – Scala is fast, due to its static typing, and it has good abstractions for programming concurrency. The combination of object-orientation and functional programming brings a whole family of new features to the JVM.
- **JRuby** – Features specific for dynamic typing; dynamic metaprogramming, dynamic dispatch/method_missing and open classes/monkey patching.
- **Groovy** – Same as JRuby, but better for tight Java integration.
- **Clojure** – Multimethods have more possibilities regarding dynamic dispatch than polymorphism. The macro system is unique, and opens up many possibilities not found in any other languages.

Using the advantages in the list above I have created table 8.1 on the following page. It gives an overview of some characteristics of tasks, and a suggestion to what programming language a DSL should be implemented in. An `x` in the table means good match, while an `(x)` means partially good match and a `-` means not recommended.

As a summary I think that Scala is the language that supports the biggest amount of tasks. Together with Java, it is the only alternative if one needs to create efficient bytecode or compiler error checking (because of the static typing + compilation). Writing concurrent applications seems to be best done in Scala or Clojure, or alternatively in Java (but with lower level concurrency abstractions). All the languages support combining DSLs, as this can be as simple as using two different libraries written in the same language together. However, some languages offer

Type of Task	Java	Scala	Groovy	JRuby	Clojure	Example
Dynamic Query Languages	-	-	x	x	-	Rails
Efficient bytecode/high performance	x	x	-	-	-	Image Processing DSL
Good concurrency abstractions	(x)	x	-	-	x	Image Processing DSL
Combining DSLs	(x)	x	(x)	x	x	Image Analysis DSL
Custom control structures	-	(x)	-	-	x	Modified host lang DSL
Compiler error checking	x	x	-	-	-	Mission Critical DSL
Error handling abstractions	-	x	(x)	(x)	x	All
Need interactive interpreter	-	x	x	x	x	All
Concise syntax	-	x	x	x	x	All

Table 8.1: Overview of tasks and suited programming languages.

better features minimizing the work needed to convert between different data structures (implicit conversions in Scala, multimethods in Clojure or open classes in JRuby).

Clojure seems like a good alternative for a number of different tasks. Memoization support and type hints may make it more suited to create efficient code than the other dynamic languages. However, the only area where it is clearly better than the other languages seems in building custom control structures and possibly removing existing ones (modifications to the host language). Functional languages on the JVM seem to have had an uprise in popularity the last years (my personal opinion based on observed Internet activity), and Clojure provides the most "pure" functional option.

The dynamic object-oriented languages (JRuby and Groovy) are best suited for building DSLs exploiting the capabilities of dynamic typing (dynamic metaprogramming, open classes and dynamic dispatching). The framework Ruby on Rails is a prime example of the possibilities these languages offer, exemplifying a super-concise syntax not matched by any other languages. If you have a use for these dynamic mechanisms, and don't require compiler type errors or high-performance bytecode, I believe these languages are good alternatives to Java.

I have found that all the languages studied provide features that let the user write much more concise and to-the-point code than what is possible in Java. This is to some extent due to the fact that all the languages support higher-order functions. Passing functions as parameters to other functions is a useful feature, that hopefully will get added to the Java programming language in a future release. All the languages, except Java, also support running as scripts or directly in interactive interpreters/REPL-environments. Writing less code for the same functionality, and being able to test it directly in an interactive environment, are both features that suggest shorter development time than what is possible in Java.

8.2 Suggestions for Future Work

As this thesis has a quite general and wide-spread focus, there are several areas that could be worth researching further and in more detail. This section is divided into subsections, each describing a separate area that deserves more attention. The topics have been identified from working with embedded DSLs, but are presented as more general features.

8.2.1 Implementing Package Templates in Scala

The package template mechanism described in [2] by the SWAT project is an advanced object-oriented mechanism that could be implemented in Scala. Some of the functionality presented in package templates is already present in Scala, in the form of traits, package objects and implicit conversions. But the dynamic capabilities enabled by the AOP mechanisms pointcut and advice are not matched, and would make an interesting task looking deeper into. Writing compiler plugins is an available and well documented feature (discussed in section 5.1.1 on page 55) that could be utilized in this work.

8.2.2 Add Support for a Dynamic Type in a Statically Typed Language

The C# language from Microsoft is a statically typed language that from version 4.0[53] adds support for a `dynamic` keyword allowing type checking to be postponed until runtime.[54] Features such as this bridges much of the gap between statically and dynamically typed languages. A suggestion of future work is to add similar support to one of the statically typed languages running on the JVM (as Scala or Java used in this thesis).

8.2.3 Implement More Metaprogramming Features in Scala

As described in section 3.12.2 on page 41 one of the major benefits of using a dynamically typed language is the powerful metaprogramming mechanisms. It would certainly be interesting to research further the possibilities of adding similar support to Scala or another statically typed language on the JVM. This task could be done in parallel with the dynamic type support described in the previous section, as dynamic typing is a key factor to some of the most exciting capabilities provided by dynamic metaprogramming.

8.2.4 Create Image Processing DSL in Clojure

Clojure is an interesting language providing a more pure functional programming model than Scala on the JVM. It would be interesting investigating further using the functional capabilities of Clojure in implementing an image processing DSL similar to the one implemented in Scala in this thesis. Although being a dynamic language, features such as memoization and type hints may make it efficient enough.

8.2.5 Implement a Complete Environment for Computationally Intensive Tasks

The main example in this thesis work has been the DSLs for image processing, statistics and charts. Joint together these form a basic foundation for an environment providing a friendly syntax in programming computationally intensive tasks.

It would be interesting working further with these ideas, focusing more on the specific domain of image processing and other tasks requiring high performance computations. A JVM based environment could potentially support several different host languages, each being leveraged for its benefits. For example using Scala and Clojure for concurrency, and Groovy for more dynamic queries (charting, reporting or similar). The building blocks requiring full control of the generated byte-code could be implemented in Java (for optimal performance), using the other languages to provide DSL syntax and other enhancements (like concurrency, querying, interactive interpreter consoles ++) on top.

[53] To be released spring 2010.

[54] C# 4.0 – Using Type dynamic: `http://msdn.microsoft.com/en-us/library/dd264736%28VS.100%29.aspx`

Bibliography

[1] Robert Atkey, Sam Lindley, and Jeremy Yallop. Unembedding domain-specific languages. In *Haskell '09: Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, pages 37–48, New York, NY, USA, 2009. ACM.

[2] Eyvind W. Axelsen, Fredrik Sørensen, and Stein Krogdahl. A reusable observer pattern implementation using package templates. In *ACP4IS'09*, Charlottesville, Virginia, USA, March 2 2009. ACM.

[3] Martin Bravenboer and Eelco Visser. Concrete syntax for object - domain-specific language embedding and assimilation without restrictions. In *OOPSLA'04*, October 24-28 2004.

[4] Krzysztof Czarnecki, John O'Donnell, Jürg Striegnitz, and Walid Taha. Dsl implementation in metaocaml, template haskell, and c++. In *Domain-Specific Program Generation*, pages 51–72. Springer Berlin / Heidelberg, 2004.

[5] Iulian Dragos and Martin Odersky. Compiling generics through user-directed type specialization. In *ICOOOLPS '09: Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 42–47, New York, NY, USA, 2009. ACM.

[6] Gilles Dubochet. On embedding domain-specific languages with user-friendly syntax. In *1st Workshop on Domain-Specific Program Development*, pages 19–22, July 3 2006.

[7] Tore Dybå, Barbara Kitchenham, and Magne Jørgensen. Evidence-based software engineering for practitioners. *IEEE Software*, Vol. 22(No. 1), Jan-Feb 2005.

[8] Burak Emir. Comparing impact of type erasure in scala and java. `http://lamp.epfl.ch/~emir/bqbase/2006/10/16/erasure.html`.

[9] Eric Evans. *Domain-Driven Design - Tackling Complexity in the Heart of Software.* Addison-Wesley, 2004.

[10] Hans-Christian Fjeldberg. Polyglot programming – a business perspective. Master's thesis, NTNU – Norwegian University of Science and Technology, 2008. `http://theuntitledblog.com/wp-content/uploads/2008/08/polyglot_programming-a_business_perspective.pdf`.

[11] Martin Fowler. Fluent interface. `http://martinfowler.com/bliki/FluentInterface.html`.

[12] Martin Fowler. Method chaining. `http://martinfowler.com/dslwip/MethodChaining.html`.

[13] Steve Freeman and Nat Pryce. Evolving an embedded domain-specific language in java. In *OOPSLA'06*, October 22-26 2006. `http://www.mockobjects.com/files/evolving_an_edsl.ooplsa2006.pdf`.

[14] Daniel P. Friedman and Mitchell Wand. *Essentials of Programming Languages.* The MIT Press, 3 edition, April 2008.

[15] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[16] Debasish Ghosh. *DSLs in Action.* Manning Publications, meap edition, 2009.

[17] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing.* Pearson Education, third edition, 2008.

[18] Philipp Haller. An object-oriented programming model for event-based actors. Master's thesis, Karlsruhe University, May 2006. `http://lamp.epfl.ch/~phaller/doc/haller06da.pdf`.

[19] Stuart Halloway. *Programming Clojure.* Pragmatic Bookshelf, 1 edition, May 21 2009.

[20] David Heinemeier Hansson. Ruby on rails website. `http://rubyonrails.org/`.

[21] Christian Hofer, Klaus Ostermann, and Tillmann Rendel. Polymorphic embedding of dsls. In *GPCE'08*, October 19-23 2008.

[22] Shan Shan Huang, David Zook, and Yannis Smaragdakis. Domain-specific languages and program generation with meta-aspectj. *ACM Trans. Softw. Eng. Methodol.*, 18(2):1–32, 2008.

[23] P. Hudak. Modular domain specific languages and tools. *Software Reuse, International Conference on*, 0:134, 1998.

[24] Paul Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, page 196, 1996.

[25] Samuel N. Kamin. Research on domain-specific embedded languages and program generators. *Electronic Notes in Theoretical Computer Science*, 14:149 – 168, 1998. US-Brazil Joint Workshops on the Formal Foundations of Software Systems.

[26] Doug Lea. A java fork/join framework. In *JAVA '00: Proceedings of the ACM 2000 conference on Java Grande*, pages 36–43, New York, NY, USA, 2000. ACM.

[27] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, Vol. 37(No. 4):pp. 316–344, December 2005.

[28] Adriaan Moors, Frank Piessens, K.U.Leuven, and Martin Odersky. Safe type-level abstraction in scala. In *FOOL 2008*, 2008.

[29] Adriaan Moors, Frank Piessens, and Martin Odersky. Generics of a higher kind. In *Proceedings of OOPSLA'08*, 2008.

[30] Dan North. Introducing bdd. *Better Software*, March 2006. `http://dannorth.net/introducing-bdd`.

[31] Martin Odersky, Vincent Cremet, Christine Rckl, and Matthias Zenger. A nominal theory of objects with dependent types. In *Proceedings ECOOP'03*. Springer LNCS, 2003.

[32] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima Press, first v6 edition, 2008.

[33] Martin Odersky and Matthias Zenger. Scalable component abstractions. In *OOPSLA'05*, pages pp. 41–57, New York, NY, USA, 2005. ACM.

[34] Daniel Spiewak and Tian Zhao. Scalaql: Language-integrated database queries for scala. In *SLE 2009*, 2009. `http://www.cs.uwm.edu/~dspiewak/papers/scalaql.pdf`.

[35] The Scala Development Team. Scala website. `http://www.scala-lang.org/`.

[36] Eivind Barstad Waaler. When is ruby more efficient than statically typed languages for developing solutions in consulting organizations? `http://tihlde.org/~eivindw/inf5500-eivindwa.pdf`, December 2008.

[37] Dean Wampler and Alex Payne. *Programming Scala*. O'Reilly, first edition, 2009.

Appendix A

Downloading and Building the Examples

Full source code for the image processing example, with statistics and charting, can be downloaded from my GitHub repository: `http://github.com/eivindw/simage`

A.1 Viewing the Code Online

The code can be navigated and viewed directly online. For example the `Image` trait with implementation and companion object is found in the file `Image.scala`:
`http://github.com/eivindw/simage/blob/master/src/main/scala/simage/structs/Image.scala`

The various pieces of code is divided into several package. Base structure starting at `simage` package:
`http://github.com/eivindw/simage/tree/master/src/main/scala/simage`

A.2 Downloading and Running the Examples

A.2.1 Prerequisites

The following tools need to be installed in order to download, build and run the examples:

- Java version 1.5 or later – `http://www.java.com/getjava/`
- Scala version 2.7.7 – `http://www.scala-lang.org/downloads`
- Git – `http://git-scm.com/download`
- Maven 2.x – `http://maven.apache.org/download.html`

A.2.2 Downloading, Building and Running

1. Download the sourcecode: git clone git://github.com/eivindw/simage.git
2. Compile code and run tests: mvn package
3. Run the example described in section 6.2 on page 62:
   (a) Go to the scripts directory: cd script
   (b) Run the example: scala −classpath simage.jar Test.scala
   (c) If everything works various generated images should be found in the script directory.