Philipp Gabler

Designing Embedded Domain-Specific Languages in Scala

A Case Study with Action Systems

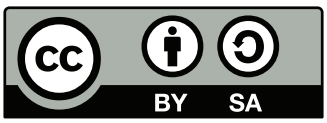
Bachelor's Thesis

Graz University of Technology
Institute for Software Technology

Supervisor: Ao. Univ.-Prof. Dipl.-Ing. Dr. techn. Bernhard Aichernig

Graz, June 2015

This work is licensed under a
Creative Commons Attribution-ShareAlike 4.0 International License.



All code samples, unless otherwise noted or cited from other sources,
are also available under an MIT license:

The MIT License (MIT)

Copyright (c) 2015 Philipp Gabler

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

A full sbt project containing many of the code samples can be
found at `https://github.com/phipsgabler/dsl-examples`.
The LaTeX source and a screen-optimized version of this document
are available at request.[1]

[1] pgabler@student.tugraz.at

Abstract

Programming languages have been and still are becoming more and more abstract, and increasingly specialized complicated applications and libraries are implemented. With that tendency, a great amount of "linguistic" flexibility is both available and needed. However, the combination of the power at hand of the programmer and the need to describe the increasingly complex systems has yet to be fully explored; this is the traditional habitat of Domain Specific Languages (DSLs).

DSLs have been in long use in the programming community. The philosopy behind them is the following: do not try to come up with a syntax so general that it can concisely express every problem (that is probably impossible); rather, define a smaller language to describe the specific problem, and embed that into a system which can combine the specific languages. In the case of *embedded* DSLs, that outer system is itself a general-purpose programming language, though preferably one which properly supports embedding small sublanguages in a convenient way.

Often, LISP has been termed such a "programmable programming language". As of today, Scala probably comes closest to this high claim, at least from the perspective of a programmer used to conventional languages. This bachelor thesis tries to explore how Scala makes it easy to write DSLs, which allow to express specific problems such that there is no additional "code noise" around them, and that the description really behaves in a transparent way, like a reader would expect it from the code.

For that purpose, in the first part an overview is given of DSLs in general and Scala's special features that help with implementing them, complemented with a description of some useful patterns for that purpose. Then, in the second part, the implementation of a specific DSL for Action Systems is described. Action Systems are a formalism originally used to formalize behaviour of distributed, concurrent systems; in the context of this work, however, they are treated more as in their interpretation as non-deterministic transition systems, which is a useful point of view for performing model-based testing.

Contents

Introduction **1**

1 Concepts: Scala and DSLs **3**
1.1 Scala: Background and Paradigms . . . 3
1.2 Overview: Why DSLs? . . . 4

2 What Syntax Can Do for Us **9**
2.1 Variants of Method Calling . . . 9
2.2 Advanced Parameter Passing . . . 14
2.3 Monads and For-Comprehensions . . . 18
2.4 Working with Literals . . . 22
2.5 Implicit values . . . 25
2.6 Macros . . . 28

3 Patterns for DSL Implementation **33**
3.1 Extensions and Rich Wrappers . . . 33
3.2 Type Classes . . . 35
3.3 Combinator Interfaces . . . 38
3.4 Objects and Modules, Mixins and Cake . . . 41

4 Action Systems and Testing: Definition & Background **47**

5 Actium – a DSL for Action Systems **51**
5.1 Building a System . . . 51
5.2 Underlying Architecture and Funtionality . . . 54
5.3 Improving the Syntax . . . 57
5.4 Extensions to the Basic Functionality . . . 64

6 Résumé: What Can Be Learned From This **69**

Bibliography **74**

A Example Programs **79**
A.1 SimpleRocket in Original Syntax . . . 79
A.2 SimpleRocketNoDSL . . . 80
A.3 SimpleRocket . . . 81
A.4 ExtendedSimpleRocket . . . 83

Introduction

Exploring Domain Specific Languages (DSLs) is a very broad topic. They have been around for a long time, and inspected in many ways, for many different purposes, in many implementations. This work therefore tries to focus on some core aspects in a limited setting. On the one hand, Scala's capabilities for writing embedded DSLs shall be investigated and summarized. This includes the inquiry of the relevant syntactic and semantic features of the language, of how and to what extent they help in the implementation of DSLs, and the examination of existing patterns for that purpose (of theoretical nature as well as occuring in practical software). This part of the exploration is mostly theoretical or based on small examples – it contains explanations about the language, about its background, and about how it can be used.

On the other hand, a somewhat larger, concrete example of an embedded DSL shall be constructed in practice. This example is a library for the definition of Action Systems, as they are used in model-based mutation testing [AJ09; Aic+14]. With this practical project, the actual development of a DSL library in Scala will be analyzed, with the aim of gaining insight into what a development process with such goals can look like (and what difficulties it might involve), into where the language helps with the embedding of a DSL and where it hampers it, and finally into how useful the previously examined language featureas are in practice – expecially, if some desireable properties are missing, or if some parts could in fact be done better with different means.

The work is structured as follows: There are two logical parts. In the first three sections, concepts and techniques for DSLs in Scala are explained. At the beginning, Section 1 gives an overall view of domain-specific languages and Scala as a programming language. It shortly introduces Scala's background and reviews DSLs in general, explaining their foundation and position in the history of programming languages, discusses advantages and disadvantages, and defines some commonly used terms. Then, Section 2 gives an intoduction into parts of Scala's syntax which are the most relevant and useful to writing embedded DSLs, includung examples of their usage. Thirdly, Section 3 shows how the previously introduced syntactic constructs can be combined into some useful patterns, which help program organization and development when working with DSLs.

Building on that, the remaining three sections consist of the description of the

practical part of the thesis: an implementation of an embedded DSL for describing Action Systems, with the working title `actium`. Initially, Section 4 gives the theoretical introduction into Action Systems, and explains their usage and special variant in model-based (mutation) testing. It follows Section 5, in which the implementation of `actium` is summarized. Special focus is layed on the design parts where the described above language features were used; notable decisions or interesting constructs used are pointed out. This section also gives an overview of the functionality and usage of the implementation. Finally, Section 6 reflects on the practical part, mentions possible improvements or further directions, and summarizes the advantages and disadvantages for this type of implementation, as opposed to other possibilities.

1 Concepts: Scala and DSLs

This part will first give a short overview of different features of Scala, followed by a basic introduction to DSLs, tracing them into the history of programming languages and concluding with the paradigm of language-oriented programming.

1.1 Scala: Background and Paradigms

Scala is a very flexible, multi-paradigm language, including a host of ideas by its inventor Martin Odersky. For one, it provides advanced object-oriented constructs: almost everything is in a class; there is a sophisticated type hierarchy; traits serve as "better interfaces", and provide very flexible ways of inheritance, like mixins; there are a lot of constructs for different aspects of class modelling, and they all can be nested. There is also a really smoothly working and consistent module concept (which is probably influenced by Odersky's experience with Modula-2). As a whole, Scala has been designed to serve as a multi-paradigm language, allowing writing programs efficiently and expressively in the small as well as in the large scale – hence also its name, stemming from *scalable language*.

Furthermore, Scala is also quite a powerful statically typed functional language, not only providing functions as first-order constructs with convenient syntax (as is minimally expected from a functional language), but also being equipped with an elaborate type system, allowing to express more things than in most other languages (like higher kinds, and certain types of polymorphism). Also, the border to object-orientation is layed out in a very uncomplicated and well-functioning way: case classes serve as class hierarchies as well as providing algebraic data types (ADTs), which can be pattern matched on, and the subtyping and parametric polymorphism systems can be fused nicely by aid of type bounds and variance annotations.

Still, it is not only this wide semantic span of paradigms that makes Scala suited for a wide range of use cases and architectorial considerations. There has been spent considerable amount of thought on a lot of larger and smaller syntactic enhancements and shortcuts, most of which allow programmers write code in a much more readable and natural way than usually. These features range from *string interpolation* and *XML literals* over *blocks* and *infix method calls* to quite unique inventions like *implicit definitions*. The subsequent sections will introduce the most useful of them for the development of DSLs, accompanied with examples of their usage and patterns of combining them well.

A few words on the level of details: surely, one cannot explain a whole language here. This text is also not an introduction to Scala. In general, it will always be tried to explain as much as possible, using plenty of examples, so that a programmer having experience in a few languages will be able to figure out what code means. However, some basic knowledge about Scala syntax is assumed, as well as background knowledge about Java, functional programming, and some aspects of semantics and type theory that are commonly discussed along with pure functional programming. In particular, this concerns concepts related to types and functions, since functional programming is considered a preliminary to all described aspects.

A more detailed foundation of Scala's syntactic features and standard libraries can be found in Odersky's introductory book *Programming in Scala* [OSV08], or the Scala language specification [Ode+14]. Java, its API, and the JVM are as well described in detail by their standardization documents [Gos+13; JavaAPI; Lin+13] (for version 7). General introductions into common functional programming techniques are the classic [SICP96] on the practical side (including untyped lambda calculus, pure and impure functional programming and a general introduction into DSL principles), and [Tho91], going into theoretical foundations and type systems (especially higher typed lambda calculi). A more thorough account of the practical type systematic approaches used in Scala can be found in [TaPL02], by which the language has noticably been influenced.

The larger examples in this part (listings on the top, marked with rules) are, in most cases, published in a Github repository[2], for the sake of better reproducibility and further study. This repository containins a compileable SBT project with their implementations, and sometimes improvements, usage examples, and other showcases. Such examples are marked and hyperlinked with an Octocat icon ( ).

1.2 Overview: Why DSLs?

Reviewing the history of computer science, as far as it is concerned with programming languages, one can distinguish two great paradigms of how computation can be expressed (these are "paradigms" more in an epistemiological than in a technical sense): *imperative* and *declarative*. This distinction can be rooted in the earliest abstractions of computation that were formulated, namely, Turing machines and the lambda calculus [VS15]. As has been early proved, both formulations are in fact equivalent; there is however a split, when practically applying these concepts, which are purely mathematical at their core.

This split has its origins in the engineering perspective of programming, which strongly influenced the evolution of programming languages. At the beginning of the usage of physical computers, all there was was machine language, which was hand-written by specialized personnel (maybe using the first assemblers, but not

[2] `https://github.com/phipsgabler/dsl-examples`

much more). Higher-level languages in these circles were believed to be a purely academic exercise and not efficient enough in practice (not an unreasonable claim at the time); it was not until the mid-fifties that they began being used. Then, between 1954 and 1957, the ancestors of all modern imperative languages, Fortran, ALGOL, and FLOW-MATIC (soon superseded by its more popular offspring COBOL) were invented and became popular [Sam72].

What was convincing about these three languages was on the one hand the Fortran compiler's ability to produce executables equivalently efficient to average hand-written machine language, but with less effort of writing – this led to the general acceptance of compilation being feasible. On the other hand, the idea behind FLOW-MATIC/COBOL (and, to a lesser extend, ALGOL) was to allow programs to be written in English language, in a descriptive fashion completely unlike the previously prevalent style of native instructions. Soon after this short phase, the number of programming languages invented started to increase almost exponentially. Languages were designed for different purposes and with different backgrounds: SQL for relational algebra queries, APL for vectorized numerical computations, C for systems programming, and so on. What is common to all languages in this tradition is their intertwined struggle between generality and expressivity – some paths evolved to what is nowadays called "general-purpose" languages (such as C++), while others became more and more specialized and even lost their abilities for programming in the whole (such as SQL).

This language family has machine language as its proto-language, and its members are all based on some form of compilation of a syntax to an underlying machine model, or, in the case of interpreted languages, executing them step-by-step on such a model. They are thus, in a sense, Turing-oriented: syntax is considered an extra layer above the actual execution on an abstract machine; and the engineering perspective mentioned above came up because concrete computers are just instances of such abstract machines.

There evolved however another language family, fundamentally different from this concept: LISP and its successors. In contrast to the languages mentioned above, LISP had never been designed as a successor to machine language. Instead, the initial idea was just a formal lambda calculus with primitive types, macros, and better syntax ("recursive functions of symbolic expressions"); the actual implementation on a computer happened only "by accident" [McC60]. To this comes the fact that initially, the main user group of the language was the AI community, finding in it a way to concisely describe their abstract concepts, regardless of underlying implementation.

While LISP over the course of its growth incorporated also imperative parts, it is fundamentally geared to the lambda calculus, with its term-rewriting style of thinking, reinforced by its own homoiconicity. In this tradition, the language lead to the invention of techniques that probably would not have been devised from the Turing perspective: these are, for example, the first-class membership of functions and continuations, dynamic typing at runtime, and garbage collection. Such concepts came into being because the language was used by people who were not using it to write machine instructions on a higher level, but who approached programming

from an almost mathematical perspecitive, wanting to specify *what* something shall be, not *how* it should be executed. In that way, LISP pioneered the idea of considering programming languages as an end in themselves.

THESE ARE THE HISTORICAL FOUNDATIONS of programming languages, summarized in short. From them, we can learn how the means of expressing intent to a machine evolved, and from what perspectives they started.

Nowadays, (most) people need not anymore be convinced that high-level languages are something positive and worth while. The challenge is to overcome the view of code as something to be solely compiled down, an intermediate, beneath form of thought, between the brain and the machine. If we instead accept code as a form of language, and take its means of expression not as syntactic niceties, but as fundamental properties of it as a system, new perspectives will come up. In that spirit, Abelson and Sussman say in *Structure and Interpretation of Computer Programs* that

> (…) we want to establish the idea that a computer language is not just a way of getting a computer to perform operations but rather that it is a novel formal medium for expressing ideas about methodology. Thus, programs must be written for people to read, and only incidentally for machines to execute. [SICP96, Preface to the First Edition]

As we have seen above, programming languages tend to oscillate between extreme poles: imperative versus declarative, and general versus specific. And as the complexity of systems and architectures grows, and their construction becomes a more professional business, all these poles have proven their value in some or the other way; but most importantly, it can be said that none of them is the key to salvation on its own. In fact, successful complex systems most often rely on a patchwork of multiple, specialized components, glued together by general frameworks of architecture. And if we bend our concept names a bit, general-purpose languages could actually be considered special-purpose for general "glue" code, and imperative code considered declarative descriptions of imperative processes.

If we so refrain from looking for "one language to rule them all", and accept the above view about the importance of languages in themselves, we arrive at a new point of view – in some form, a new paradigm: *language-oriented programming* [Dmio4]. Its philosopy is exactly the just described manner of organizing systems: splitting them in smaller, coherent parts, and program them out using the respective *domain specific language*, which is a language as close as possible to the terms and concepts of the specific part.

THIS LANGUAGE-ORIENTED PARADIGM and its main constituents, DSLs, have actually been around for some, often under different names, or in the context of other methodologies. For example, the programming approach suggested in [SICP96] is there called *metalinguistic abstraction* and proposed as a rather general approach to problem solving; but then, there have always been used small configuration or mini-languages for systems such UNIX (like the sed program), which are not backed by

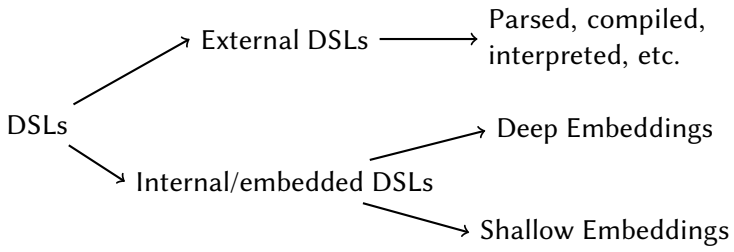


Figure 1.1: Taxonomy of terms specifying the different variants of Domain Specific Languages.

such a general idea, but just evolved as practical. Moreover, there is range of highly specific languages to help with specialized tasks, including the already mentioned SQL, the grammar specification language of ANTLR or other parser generators, and LaTeX. We can also regard special syntaxes for specific purposes as a kind of DSL, like Matlab's matrix literals and special operators, or the simulation statements of hardware description languages such as Verilog.

A summary of the advantages of using DSLs is given in [Spi01], which also lists notable patterns found in the development process of them. [MHS05] offers an overview of decision, analysis, design, and implementation phases of DSL development, from a more methodological view point, altough the work focuses more on external languages. Case studies for DSL development are [Wil04] with a lot of more abstract "lessons learned", and [HIW10], which compares two implementations of the same DSL in Python and Scala with respect to their "linguistic elegance" (and even contrasts multiple variations in Scala).

WITH ALL THIS VARIETY of declarative and in some sense domain-specific languages, we could try to classify them (see also Figure 1.1). The most prominent and used distinction for DSLs is that between *external* and *internal* DSLs. Thereby, a DSL is called internal if it is used as part of another, more general programming language (either through special syntax, or provided by a library), and called external, if it is executed by a dedicated "evaluator" different from the main implementation language (which can be anything from a compiler over an interpreter to a DBMS).[3]

While external DSLs could arguably be differentiated in arbitrarily complex ways, some basic further discrimination is practical for internal ones. For one, there are the two said possibilities of having an internal DSL provided by the language or via a library. The latter variant in the following will be referred to as *embedded*[4]. It is embedded DSLs that this work focuses on, since they are in a sense the most "language oriented" – using the features of a language to extend itself.

[3] Again, SICP tries to be smarter than and blur this binary distinction: "The evaluator, which determines the meaning of expressions in a programming language, is just another program." [SICP96, Chapter 4]

[4] As far as is known to the author, there is no generally accepted convention for a clear distinction between the terms "embedded" and "internal", but this usage seems useful.

Within the range of embedded DSLs, there is a further differentiation to be made: that of *deep* and *shallow* embeddings. This distinction concerns the evaluation of the embedded syntax: in a shallow embedding, the provided language constructs are merely combinators for some already-existing objects in language, for example, they may simply be stacking functions in an intricated way – but evaluation is not different from the normal language behaviour. In contrast, a deep embedding uses an internal AST, or at least some symbolic representation, together with a custom evaluation function. The latter has the advantage that there is often more semantic flexibility involved, and that there might be multiple evaluation functions provided ("backends"); however, this usually comes with greater complexity and runtime or space costs. A survey and comparison of both approaches can be found in [GW14], which uses a Haskell example.

2 What Syntax Can Do for Us

This section should show what makes Scala a particularly good choice to write DSLs in. It focuses on syntactic features, using examples to explain their usage and motivation, and also introduces some additional small tricks related to them, as well as useful idioms that fit nowhere else. The features described here are selected because of their suitablity for building *control* or *data abstractions*: they allow to build interfaces with structures to compose data and behaviour in an expressive way, without burdening with too many boilerplate declarations.

2.1 Variants of Method Calling

Scala, being based on a platform initially constructed for object-oriented programming, and being designed with full utilization of this paradigm in mind, naturally has a notion of *methods*. It also makes plenty of use of them, and in fact, all behavioural functionality of the language can be reduced to method calls. However, unlike other object-oriented languages of similar kind, notably Java, Scala allows for a much higher syntactic flexibility of these, while under the hood still working with the same mechanisms. These syntactic flexibilities mainly appear in the form of *operators* and *infix notation*.

In the this work, the term "operator" shall always mean a function whose name consists of non-alphabetic symbols and that is written infix (if binary) or pre- or postfix (if unary) – the same terminology as used in the language specification. With "infix notation", the style of calling a "normal" method without dots and parentheses, as in `v foo x` instead of `v.foo(x)`, will be meant. To distinguish "regular" method calls from them, often the term "dotted call" will be used. It is worth noting that most things being said here for methods hold as well for type constructors (i.e., class names like `::`), but these are not further investigated.

Overwriteable operators are an idea that has been around for some time in other languages, although mostly in quite restricted form. Often, there are ways to redefine arithmetic and boolean operators, and comparisons. The former is what one usually needs when implementing numeric data types, such as matrices, that are not part of the standard libraries; the latter serves mainly for providing custom equality and orderings with syntactic integration, which is quite often desirable. There is of course also some "bending of the rules", like the often-appearing string

```
sealed trait Expr {
  def ||(other: Expr) = Or(this, other)
  def &&(other: Expr) = And(this, other)
  def unary_! = Not(this)
}

case class Var(s: String) extends Expr
case class Or(a: Expr, b: Expr) extends Expr
case class And(a: Expr, b: Expr) extends Expr
case class Not(a: Expr) extends Expr

implicit def stringToExpr(s: String): Var = Var(s)
```

Listing 2.1: A small DSL to represent symbolic propositional formulae. The implicit conversion in Line 12 allows to leave out the `Var` constructor; such conversions are explainened in Subsection 2.5.

concatenation with +, or C++'s stream piping with << and >>. From all these use cases we see that programmers have a tendency to use operators where they seem the most natural (like in mathematical contexts), or where they seem to be able to express themselves better than with dotted method calling style, because of shortness or since their infix style may resemble the logical or data flow better.

Some languages with a different background, mostly Haskell and Agda, take an opposite and quite radical approach: operators there are just names for functions, which by default can be written in infix notation (similarly LISP, though it exclusively uses Polish Notation). This functionality has lead to a style of combinator libraries in them (like Haskell's `parsec` parser combinators[5]), which can be quite expressive and concise at the same time (using these in Scala is also discussed in Subsection 3.3). Scala, being partly influenced by these languages, provides an intermediate way, but almost as expressive: namely, method names are allowed to contain almost all unicode characters; and methods containing only non-alphabetical characters can be written inline. These infix calls are recognized and converted into standardized textual and dotted forms:

```
xs ++ ys  ≡  xs.++(ys)  ≡  xs.$plus$plus(ys)
```

As can be seen, the intermediate form of using the symbolic name of an operator in a dotted call is also allowed.

This translation is quite mechanical, and it does really not add any new functionality. It can, however, massively improve the readability of certain types of programs. For example, the logic DSL from Listing 2.1 allows to write a logic formula like this:

```
"a" || "b" && !"c"
```

instead of the following, which would be necessary in Java:

```
new Or(new Var("a"), new And(new Var("b"), new Not(new Var("c"))))
```

This example raises the question of how operator precedence behave. Can we be sure that when writing `!"a" && "b" || "c"`, we do not, against our expectations,

[5] `http://hackage.haskell.org/package/parsec` (visited on 2015-06-04).

end up with `Not(And("a", Or("b", "c")))`? It turns out that we can, at least in this case. But determining precedence of operators (or conversely, assigning the right precedence to an operator when defining it) can be a bit awkward in Scala. That is because, where other languages allow one to explicitly specify precedence and associativity of operators, in Scala these are determined solely by the name (i.e., the symbols) of the operator in question. This hard-wired convention is defined in the language specification [Ode+14, Chapter 6.12]. The logic behind is basically the following:

1. Precedence is determined by the first symbol
2. There are fixed values for the operator symbols occuring in practically any language (`|^&=!<>:+-*/%`, in increasing order); all other non-alphabetic symbols equally bind stronger
3. There are exceptions for assignment operators, ending in `=`
4. There are exceptions of these exceptions for comparison operators
5. The default associativity is left, unless the operator ends with `:`

It is also possible to define unary prefix and postfix operators. Postfix operators are not really a special case, as they fit into the dotless method calling style explained below; prefix operators, however, as the negation operator `unary_!` in Line 4 of Listing 2.1, deserve to be mentioned separately. These are declared by defining a method with name `unary_`$\circ$, where $\circ$ is the name of the actual operator – currently, however, the allowed symbols are constrained to `+-!~`.

There is another question concerning operators, which is that of integrating them into existing types when allowing mixed-type applications. For example, given a matrix class `Matrix`, it is possible to implement `*` as a method of that class, such that for `m: Matrix`, the expression `m * 2` typechecks; however, the expression `2 * m`, looking equally well-typed, cannot be provided simply, since the `*` operator of `Int` does not have any overload taking `Matrix` arguments, and `Int` cannot be just be "patched". The usual workaround for this problem is to provide an implicit conversion from `Int` to a wrapper class having the desired method; Subsection 2.5 shows how that can be achieved in an "invisible" way, essentially reintroducing the expression `2 * m` into the language without having to actually change any class.

As soon as we realize that operators can by reduced to simple method calls, we could also go the other way round: using method names, like they were operators. This sounds like a promising idea: expressions like `xs contains "bar"` become possible in this way. And in fact, such "dotless" calls are a very commonly used style in Scala. They can help readablility in different ways: for one, less-speaking method names like `map` or `withFilter` can be applied like this in a pipeline-like fashion. The following is a quite typical example, taken from the practical part of this work:

```
a.statements map (_.pretty) mkString ","
```

Or, even more readable (at least for the initiated):

```
1 class ExampleSpec extends FlatSpec with Matchers {
    "A Stack" should "pop values in last-in-first-out order" in {
      val stack = new Stack[Int]
      stack.push(1)
5     stack.push(2)
      stack.pop() should be (2)
      stack.pop() should be (1)
    }

10  it should "throw if an empty stack is popped" in {
      val emptyStack = new Stack[Int]
      a [NoSuchElementException] should be thrownBy {
        emptyStack.pop()
      }
15  }
  }
```

Listing 2.2: Example of a ScalaTest DSL for writing unit tests in a natural-language-like specification. This example is taken from `http://www.scalatest.org` (visited on 2015-05-21) and slightly modified. Also note the anaphoric use of `it` in Line 10.

```
require(nats forall (_ >= 0), "List contains negative numbers")
```

which defines a precondition on a value `nats` (a list of `int`s), by declaring that all its members must be nonnegative (otherwise exiting with an error message).[6]

This style of calling is often exploited in DSLs resembling phrases of natural language; such usage can be pushed quite far, as Listing 2.2 shows. When defining such an interface, a standard trick is to construct wrapping objects closing over the relevant parameters. These objects can then provide methods in an order and wording close to a natural utterance; even "no-op" methods, like the `be` in the example, can then easily be included and interpreted. As an example, we could define a kind of "filter pipeline" for lists in the following way:

```
def map[A, B](f: A => B) = new {
  def over(xs: List[A]) = new {
    def skipping(p: A => Boolean) = xs filter (p andThen (!_)) map f
  }
}
```

In this case, `map` closes over a function and creates an anonymous object with a method `over`, which closes over a list and returns another object with a `skipping` method, in which the closed-over parameters are finally used to perform a filter-and-transform operation over the list. These methods can then be used in such a style:

```
scala> map ((_: Int) + 2) over List(1, 2, 3, 4, 5) skipping (_ < 3)
res0: List[Int] = List(5, 6, 7)
```

Unfortunately, in this case, an explicit type annotation is necessary in the first lambda expression, because of Scala's local type inference; however, in DSLs, parameters will often be of fixed custom types, relieving this effect. The example also uses

[6] Example from `http://www.scala-lang.org/api/current/#scala.Predef$` (visited on 2015-06-02).

`reflectiveCalls` to allow using anonymous types for shortness; in real code, every returned interface should be explicitly given by a trait. Using traits for this style also has the advantage that different "paths" of calling can be restricted by defining methods only as extensions on mixin combinations, and not directly on the traits themselves.

There are some additional syntactic features to be mentioned in this subsection. These concern the ability to define getters and setters. It is a very common style in object-oriented languages not to expose fields of a class directly, but to provide instead methods like `void setFoo(T value)` and `T getFoo()` to encapsulate the underlying value. This also has the advantage that invariants of the internal state can be ensured using pre- and postconditions, and more fine-grained access control is possible (like disallowing setting). On the other hand, such methods are an overhead: they are not standardized, and introduce the look of a method call where logically `foo = t` should be enough.

Other languages, such as Python and C#, do have the ability to implement overloaded getting and setting of field assignment syntax, (with the `get`/`set` statements in C#, and the `__getattr__`/`__setattr__` magic methods in Python). This is very commonly used to ensure invariants on the values, but also to provide "virtual" fields, which instead of wrapping a variable are computed on-demand from other information. Now, when we already have overloaded member assignment, there naturally comes the desire to overload indexed (array member) assignment, too. While in C# and Python, it is possible to overload `()` and `[]` separately, in Scala, these are both unified under `()`, since square brackets are already in use for type arguments.

To examplify Scala's syntax for all this, consider Listing 2.3. It contains a small mutable wrapper around a (not further specified) immutable dictionary implementation, and has four methods: for updating and retrieving dictionary entries, and for getting and setting the maximum capacity of the dictionary. Retrieving an entry is implemented using the method `apply` (Line 13). This method is always used when an object is called like a function: in this case, given `d: MutableDict[Int]`, we could say `d("foo")` to get out the value at `"foo"`. The application gets translated to `d.apply("foo")`, which resembles the fact that indexing is not logically different from calculating a function value. Because of this, `apply` is also the whole magic that lies behind the `Function` interfaces of the standard library.

Consistent with this, we can also write `d("foo") = 42`, to update that entry. This is achieved by providing `update` (Line 5), which is similar to apply, but takes as an additional parameter the new value, and returns `Unit` (since updating is only a side effect; it is also common to return the updated dict here, to allow method chaining). In this case, the updating operation also keeps track of the size, and ensures that the dictionary does not exceed its maximal capacity. Both `update` and `apply` can in principle take multiple parameters of any types, which allows to implement complex, multi-dimensional access, if necessary.

The examples for field assignment are Lines 15 and 16. The getter of `capacity` is

```
class MutableDict[T](private[this] var c : Int) {
  private[this] var dict: DictImpl[T] = DictImpl.empty
  private[this] var size: Int = 0

  def update(key: String, value: T): Unit = {
    if (dict.get(key).isEmpty) {
      require(size < capacity, "Capacity exceeded!")
      size += 1
    }
    dict = dict.updated(key, value)
  }

  def apply(key: String): Option[T] = dict.get(key)

  def capacity: Int = c
  def capacity_=(newCapacity: Int): Unit = {
    require(newCapacity >= size, "Capacity too small!")
    c = newCapacity
  }
}
```

Listing 2.3: A mutable wrapper around an immutable dictionary type `DictImpl`, ensuring the size to stay below a fixed capacity. The capacity can be reset, but only increasingly – it always has to be bigger than the current actual size, which is ensured in the setter as a precondition.

just a parameterless method with the same name, which is not a special syntactic feature. But it is also possible to accompany any such a method, having name x, with a setter function, called x_=, which is used for assignment syntax; given such a method, `obj.`x `= y` will be translated to `obj.`x`_=(y)`. In the above case, `capacity_=` has the additional effect of ensuring that the newly set capacity cannot be less than the current size.

2.2 Advanced Parameter Passing

Scala has a lot of syntactic features that make it easier to write DSLs in a very flexible, but at the same time intuitive way. Since we now have seen how conventional calling chains can be broken up using infix notation, we can complement this advantage with various ways of "advanced parameter passing". Combining the techniques of both kinds leads to a style that looks a lot like a were native syntax, but is actually only syntactic sugar for various forms of method calls.

One important concept of advanced parameter passing is that of a *block*. With blocks, Scala generalizes such notions as "bindings", "local closures", or similar constructs. The concept is quite simple: an expression of the form

```
{
  stmt_1
  ⋮
  stmt_n
```

```
  expr
}
```

creates a local scope (like in other curly-brace languages), which, when evaluated, results in the sequenced side effects of the $\texttt{stmt}_i$; but at the same time, the whole block represents an expression soleley consisting the value of the last expression `expr` (modulo bindings introduced). This is, first of all, useful for defining local variables and occasionally interleaving side effects (e.g., for debugging):

```
val x = foo {
  val y = some + complicated(nested, expression)
  println("Debug: " + y)
  y
}
```

which is denotationally equal to

```
foo(some + complicated(nested, expression))
```

but also will print out the intermediate value of `y`.

Above, we can also observe a second important syntactic feature: unary function calls, which usually require the argument to be put in parentheses, can be called directly on a block (`foo` here would just have a type like `T1 => T2`). This style allows the user to program against a library like it consisted of language statements – given that the library was intended to be used as such. Still, often this possibility is not enough to provide a natural interface, because the default evaluation order of application does not correspond to what is expected by the reader of the code. How this can be resolved is examplified in the next subsection.

Blocks can also be used as result of lambda expressions, relieving the language from providing binding constructs such as `let` forms and generally unifying the style of programs. In the following, we define a function `f`, which calculates `x` as above, but with `nested` provided through an argument:

```
val f: (A => B) = nested => {
  val y = some + complicated(nested, expression)
  println("Debug: " + y)
  foo(y)
}
```

In most programming languages, parameters passed to a method are evaluated when the method is called. That is, in a statement like

```
foo(bar(), y + z, { log("hi!"); foo })
```

every one of the arguments will show its side effects at the time `foo` gets called. As mentioned, however, this will sometimes be a behaviour that we want to avoid. To illustrate the concepts to resolve this, in the following an implementation of "delays" or "thunks" will be used – a means to encode lazy evaluation into a strict functional language, having long been used in Scheme [SICP96, Chapter 4.2] and going back to the examination of lazy evaluation orders of lambda calculus.

```scala
class Delayed[+T](thunk: Unit => T) {
  private[this] var cached: Option[T] = None

  def force: T = cached match {
    case Some(value) => value
    case None => {
      val value = thunk(())  // evaluation happens here
      cached = Some(value)
      value
    }
  }
}
```

Listing 2.4: Scheme-inspired `Delayed` implementation, using a function from `Unit` and and a mutable variable for caching.

The original idea is to simply represent a thunk of type `T`, that is, a value to be caluculated by need, as a function of type `Unit => T`, where `Unit` is the trivial type containing only one value, `()` (also pronounced "unit"). This way, instead of passing around some `T`, which of course would be evaluated immediately, we can hold a function which can compute the desired `T` later, if we actually need its value (this is usually done using a function called "force"). Once evaluated, the value is also cached, so that expensive evaluations happen at most once. One way of translating this into Scala would be the one shown in Listing 2.4.

Here, `private[this]` is a Scala specific scope modifier marking `cached` as *object private*: that way, not even objects of the same class can access the field. This restriction is even stronger than what is possible in most other object-oriented languages, but serves an excellent purpose of hiding mutable state, so that from outside, we can still speak of a purely functional (thus immutable) object. To compare with C++, this has a similar spirit to marking fields used for caching `mutable`, but leaving the accessors `const` – holding the promise of referential transparency lies in the programmers responsibility, but can improve performance if used wisely.

Still, the above implementation leaves room for improvement. To construct a `Delayed` value, one needs to manually call the constructor on a an anonymous function:

```scala
val d1 = new Delayed((_unit) => {
  println("heavy computation")
  2
})
```

We even need to name the superfluous `Unit` argument. Additionally, while the construction as shown is practically immutable from outside, it would be desireable to even get rid of such small traces of impurity.

Fortunately, both aspects can be improved by the help of the language. For this, we can use two new features: *call-by-name* functiona arguments (indicated by the `=>` before the type) and *call-by-need* values (`lazy val`). Using them, we end up with the code from Listing 2.5, which is shorter, and at the same time purer and more easily usable. Using that, we can write the following:

```
class Delayed[+T] private (thunk: () => T) {
  private[this] lazy val cached: T = thunk()
  def force: T = cached  // evaluation happens here implicitly
}

object Delayed {
  def delay[T](delayed: => T) = new Delayed(() => delayed)
}
```

Listing 2.5: Improved `Delayed` implementation, using Scala's laziness and block constructs. The `delay` "factory" is declared inside the companion object, while the default constructor of `Delayed` is set private.

```
val d2 = delay {
  println("heavy computation")
  2
}
```

The changes introduced in this improved version are to use Scala's laziness constructs: the `cached` value is now a `lazy` reference to the evaluation of the thunk, and `force` simply returns the value directly – which in turn evaluates the thunk. The subtle difference between call-by-name arguments and `lazy` is the caching behaviour: while both are evaluated non-strictly, a `lazy val` is stored, as soon as it has been evaluated, while by-name parameters are reevaluated every time (internally, they are converted to `def`s like the `thunk` representation). It is the interplay of these two strategies that makes this way of argument passing so useful.

Furthermore, a factory method `delay` is provided, which takes as an argument an unevaluated `T` and wraps it into the old constructor taking a function, totally hiding the underlying implementation – consequently, the actual constructor is now marked `private`. This unevaluated `T` is intended to be provided as a block, a form in which form the code now barely looks different from a native language construct.[7]

As can be seen in the last example, it is possible for functions to be called on blocks (with braces), which makes them look like they were language statements. In fact, every application to a single argument can be replaced by a block. When designing APIs in Scala, this possibility is regularly exploited, and also occurs frequently in the standard library. But often, we also want to simulate a parametrized statement, like in in this example:

```
repeat(5) {
  println("hello!")
}
```

which would just print out `"hello!"` five times.

Defining methods with this interface is possible as well, if we take into account

[7] Thinking this example further, we could just get rid of `Delayed` at all and only use call-by-name and lazy values, since they in fact use the same techniques under the hood. The purpose of this was, however, to show the possibilities of defining new syntax-like methods, not of seriously implementing behavioural data structures.

```
def repeat(n: Int)(block: => Any): Unit =
  if (n > 0) {
    block
    repeat(n - 1)(block)
  }
```

Listing 2.6: Simple combinator for repeating an action n times, using a curried parameter list. The `block` needs to be passed by name, as it is evaluated in each iteration for its side effects.

the more accurate description of the "block application" syntax: namely, that the *last single application* of the parameter lists of a method can be replaced by a block. From this it follows that we can use *curried methods* to write "pseudo-statements" with parameters. An implementation of the example `repeat` function is shown in Listing 2.6 (which also is another example of call-by-name).

Curried methods are methods with multiple parameter lists, written like this:

```
def fn(x11: X11, ..., x1k1: X1k1)...(xn1: Xn1, ..., xnkn: Xnkn): Y
```

They are included into the language to simplify partial application, of which the above use case is an instance. Partial application is the mechanism allowing to "fix a parameter" of a function; that means, if we have `f: (A, B) => C` and some given `a: A`, then we could write `g: B => C = (b => f(a, b))`. But this can be expressed simpler, if `f` is curried: from `f2: A => B => C`, we can make `g2: B => C = f(a) _`. (That these are always interchangeable is a consequence of the fact that there is an isomorphism (to be exact, an adjunction) between all types `(A, B) => C` and `A => (B => C)`.)

This does not by itself reveal immediate usefulness (especially since there has to be included a "dummy" `_`), but is a practical feature when dealing with higher-order functions. Additionally, besides the described syntactic trick, there are other cases in which curried functions are helpful: e.g., they can help with type inference, since (speaking simplified) for each parameter list, there is a separate type unification step, after which some type parameters can be fixed (this is often exploited in multi-parameter polymorphic higher-order functions such as `foldRight`/`foldLeft`). Similarly, implicit method parameters (explained later in Subsection 2.5) are a form of curried parameters, and the implicit parameter list is resolved separately from the other parameters.

2.3 Monads and For-Comprehensions

In the previous subsection, a (rather primitive) implementation of delayed values was shown. This implementation was already able to do what is expected, and provides an intuitive construction interface; however, in its minimal state, composing different delayed actions quickly becomes tedious. Imagine a scenario in which we have ways to query a remote database and a web API, both in a delayed way (using something like the above implementation):

```
def getUrl(id: String): Delay[String] = delay {
```

```
  dbConnection.getId(id).address
}
def getFromApi(url: String, path: String): Delay[String] = delay {
  webRequest(url).get(path)
}
```

Now, if we want to combine these requests, we would have to do the following:

```
val getInformation(id: String): Delay[String] = delay {
  val address = getUrl(id).force
  val info = getFromApi(address, "/info").force
  "Address " + address + " has info " + info
}
```

Here, `force` needs to be called twice inside the `delay` block to unwrap the results of the nested delayed calls, which are then combined and wrapped again.

This was just an easy case. If there happen to be more layers, or we have to interleave other operations such as validations of data or error handling, the repeated nesting of `delay` and `force` does not anymore keep its conciseness, but starts to look awkward and even becomes a source of error and confusion (a result which is known as "callback hell" in the JavaScript community).

The complication introduced by combining this style of operations, involving types of the form `T => M[T]` where `M[_]` is a kind of "context" (in this case, the delay of values), has been known for long; and there is a pattern to resolve it, based on the observation that most of the time, the `M[_]` used will be a *monad*.

A monad is a category-theoretic construct on functors, requiring them to be able to be transformed in certain ways according to some laws. A theoretically founded treatment, which still refers to the programmer's point of view of "context containers", can be found in [Mog91]; but in essence, a monad can be described as a parametrized type `M[_]`, with a "constructor function" of type `A => M[A]` for all A (in the following called `pure`), such that on every `M[A]` there are methods

```
def map[B](f: A => B): M[B]
def flatMap[B](f: A => M[B]): M[B]
```

subject to the following conditions:

1. `m map (x => g(f(x))) ≡ m map f map g`
2. `pure(x) flatMap f ≡ f(x)`
3. `m flatMap (pure _) ≡ m`
4. `(m flatMap f) flatMap g ≡ m flatMap { x => f(x) flatMap g }`

where `m` is a monadic value of type `M[A]`, and `f`, `g` are "actions" of types `A => M[B]` and `B => M[C]`.

It is the type of `flatMap` that allows to sequence actions of types like `A => M[B]` in a concise form. The "purpose" of the monad laws thereby is to ensure that using `flatMap` behaves sensibly with respect to the structure of the underlying type; they are also the responsible for the ability to rearrange the below-mentioned comprehension syntax in a meaningful way.

```scala
sealed trait Delayed[+T] {
  import Delayed._

  def force: T

  def map[U](f: T => U): Delayed[U] = delay {
    f(this.force)
  }

  def flatMap[U](f: T => Delayed[U]): Delayed[U] = delay {
    f(this.force).force
  }

  def foreach(f: T => Unit): Unit = f(this.force)
}

object Delayed {
  def delay[T](delayed: => T) = new Delayed[T] {
    private[this] lazy val cached: T = delayed
    def force: T = cached
  }
}
```

Listing 2.7: Full example of the `Delayed` implementation, with monadic functions. Additionally, the `Delayed` class was replaced by a trait, of which anonymous instances are created by the factory `delay`.

The point of using monads is that Scala provides so-called *for-comprehensions* for them for free, for any type implementing the necessary interface (usually, `map` and `flatMap`; possibly also `foreach` and `withFilter`). These comprehensions generalize the `for`-syntax of the language to arbitrary monadic types, allowing one to write nested (flat)mapping of functions in a linear, somewhat imperative-looking way. As an example, in Listing 2.7, the `Delayed` class of the last subsection has been rewritten to support a monadic interface. Using the comprehension syntax with this implementation, the previously mentioned example can be rewritten like this:

```scala
val getInformation(id: String) = for {
  address <- getUrl(id)
  info <- getFromApi(address, "/info")
} yield ("Address " + address + " has info " + info)
```

This comprehension gets translated as follows:

```scala
getUrl(id)
.flatMap(((address) => getFromApi(address, "/info"))
.map((info) => "Address " + address + " has info " + info)
```

Desugaring for-comprehensions happens before type checking and does not ensure any behavioural guarantees, so it practically amounts to duck-typing of whatever values are used. The exact way of translating is documented in the language specification [Ode+14, Chapter 6.19].

In fact, the `Future` type of the standard library (`scala.concurrent.Future`) is not so much different from this version of `Delayed`, except that it runs thunks not later on

demand, but asynchronously to the current context (an implicit `ExecutionContext` is passed with most methods on futures). Futures, too, have a factory called `Future` and implement the monadic interface, and it is considered good style to use the available combinators implemented on it, instead of nesting `Future` calls and blockings.

The type of behaviour that is provided by monads can be compared to SQL statements, which is also the reason that for-comprehension are used mostly for collections (for comparison, the C# equivalent of for-comprehensions, called LINQ (Language Integrated Query)[8], even uses SQL keywords like `from` and `select`). The type of `flatMap` allows to express "nested selection" and "joins" on arbitrary "containers", with whatever semantics make sense for them: sequencing, nondeterminism, delayed continuation, and so on. For example, we can provide the following combinator:

```
def join[A, B](a: Iterable[A], b: Iterable[B]) = new {
  def by[C](f: (A, B) => C): Iterable[C] = for {
    x <- a
    y <- b
  } yield f(x, y)
}
```

which can be used like this[9]:

```
scala> join(List(1, 2, 3), List('a', 'b', 'c')) by ((_, _))
res0: Iterable[(Int, Char)] = List((1,a), (1,b), (1,c), (2,a), (2,b),
  ↳ (2,c), (3,a), (3,b), (3,c))
```

(Note also the trick of introducing the intermediate object with a `by` method, which is called in infix notation!)

The same code, replacing `Iterable` with `Future`, can be used to get a combinator with "await both" semantics for Futures; in fact, it would make some sense for any monad. For this reason, monads are often subsumed under a type class (Subsection 3.2), and there are libraries like Scalaz,[10] providing a range of general methods and combinators for all monads (and many other general classes). Another notable monad, which is not immediately appearent as such, is the `Parser` type of parser combinators (cf. Subsection 3.3), for which the monad provides conditional and error-preserving sequencing of parsing functions.

There is one additional thing to say about the use of the term "monad" or "monadic" in Scala or other languages supporting syntactic sugar for them: sometimes, for a type, the monad syntax (i.e., the comprehensions) still makes sense,

[8] Monad comprehensions as "generalized for" have found their way not only into C#; the idea, initially implemented in Haskell's *do-notation*, is also present in F#'s *computation expressions*, which generalize the concept even more by the use of continuation-passing techniques [Sym12, p. 62].

[9] Calling this function will lead to a warning signalling that this only works using the compiler flag `reflectiveCalls` Using a trait instead of an anoymous structural type would be considered better style, and is probably safer, but requires more setup than this example. The function also does not work really well with the collections hierarchy, as it does not properly preserve container types.

[10] `https://github.com/scalaz/scalaz` (visited on 2015-05-16).

even if the monad laws are not strictly fulfilled. For example, `scala.util.Try` from the standard library does not behave the same on both sides of the second above-mentioned law, because of its special exception-capturing behaviour. Or, types for random variables, which are monads too, do not fulfill the laws by conventional equality; instead, they only guarantee the left and right hand sides of the equations to have identical probability distributions. But in all these cases, using comprehension syntax is nontheless useful and adequate enough to not prohibit it. For that reason, also such types are still commonly called "monadic".

2.4 Working with Literals

When developing custom data types, we want them to be usable as naturally as possible, especially in the context of DSLs. This includes working with values of them in an intuitive and integrated way – namely, handling custom literals should not be different from handling "primitive" values, and ideally, custom types should be able to be written exactly as natural in their domain. Scala also provides some means to achive this: first, it has the possibity to provide custom *string interpolators*; and furthermore, new *extractors* can be defined for arbitrary types, allowing to create new pattern matching ability without having to define case classes.

String interpolators are, for one, a way of providing readable constructors of data types through strings, essentially by decomposing normal string literals interleaved with arbitrary expressions into a function application. There are three interpolators provided by the Scala standard library: `s`, `f`, and `raw` [Sue]. The first one does not do much more than providing syntactic sugar for string building (using `toString`):

```
val x = 123
val y = (true, List(1, 2, 3))
val str = s"Number: $x, boolean: ${y._1}"
```

Here, `str1` will result in the string `"Number: 123, boolean: true"`. As can be seen, the interpolator escapes identifiers starting with `$`; and in general, arbitrary block expressions can be written inside curly braces following a `$`. So, continuing the above example, we can write an expression like

```
s"""The sum is ${if (y._2.sum % 2 == 0) "even" else "odd"}"""
```

which evaluates in this case to `"The sum is even"` (the quotes inside braces are automatically "escaped", since desugaring happens independently before literal parsing). The other two standard interpolators work in a similar way, but additionally allow `printf`-style format specifications or ignore escape sequences (like `\n`).

More interesting, however, is the fact that we can write our own interpolators. Every call to an interpolator of the form `intp"some string"` gets rewritten by the compiler to a method call on `StringContext`, which is essentially only a container for an array of strings: `StringContext("some string").intp`. If there are parameters

in the string, these get passed to the method: `intp"using $x's value"` becomes `StringContext("using ", "'s value").intp(x)`.

This rewrite rule is extensible insofar as we can add methods to `StringContext` through implicit conversions (see Subsection 2.5). These allow to (syntactically) extend classes without inheriting from them. Using this technique, one can write new interpolators using a wrapper like the following:

```
implicit class Bla(sc: StringContext): AnyVal {
  def foo(params: Any*) = fooImpl(params)
}
```

This can then be applied to any string literal with an arbitrary number of arguments, like `foo"x $a y $b z"`, reducing to

```
Bla(StringContext("x ", " y ", " z")).foo(a, b)
```

While it is common and often useful to allow arbitrary parameters (`Any*` is internally represented as a `WrappedArray`), an interpolator can in principle take any number of concretely typed parameters. Since the interpolator call is translated to an equivalent method call anyway, incompatible or insufficient calls are automatically refused by the type checker. And while the `s` interpolator is relatively trivial (it just adds the parameters and the string chunks), a more complex interpolator will often involve parsing the string which is interpolated, and even introduce some extra internal syntax to modify the behaviour of the passed interpolants.

String interpolators are widely used for easier construction and deconstruction of text-based data. Prominent examples are constructors for JSON [Voi13], and recently quasiquotes, which are used in macro programming and allow the construction of Scala ASTs without having to construct them "by hand" [Sha; SBO13]. With the help of macros, quasiquotes can also be pattern matched on. Even the XML literal syntax of Scala is planned to be replaced by interpolators in future versions [Ode15].

ANOTHER IMPORTANT USE of literals, apart from constructing them, is deconstructing them – that is, accessing their values. For that purpose, there is in principle already a solution: just using members returning whatever is logically contained in a class. Now, since Scala has the feature of case classes, which allow pattern matching, these will often be preferable; however, not every datatype will be written as a case class. The most obvious example for that would be a class implemented in Java. In other cases, the underlying implementation might be much more complicated than the logical form of the concept, and is not wanted to be exposed; this would e.g. happen in a set implementation. So, there arises the need for being able to define custom patterns.

Additionally, as soon as we have a means of introducing new patterns, we can do even more things than were initially looked for: with newly definable patterns, we are also able to introduce synonyms to other patterns (or means of deconstruction), or we can provide more content specific matching of data, deconstructing objects in a semantically richer way.

The way Scala allows defining custom patterns is through *extractors*. An extractor

```scala
object || {
  def unapply(e: Expr): Option[(Expr, Expr)] = e match {
    case Or(a, b) => Option(a, b)
    case _ => None
  }
}

object ! {
  def unapply(e: Expr): Option[Expr] = e match {
    case Not(a) => Option(a)
    case _ => None
  }
}
```

Listing 2.8: Extractor objects for disjunction and negation, as defined in Listing 2.1. Conjunction is analogous to disjunction. Note that, since there are no unary tuples in Scala, `unapply` for `!` only needs to return `Option[Expr]`.

is an object with an `unapply` method, taking as parameters the value that should be matched, and returning an `Option` of a tuple of the possibly extracted values. The `None`-ness of the `Option` signifies whether the match succeeded. As an example, look at the extractors in Listing 2.8, which accompany the DSL for logic expressions from Listing 2.1. These definitions (plus the left-out one for `And`) allow to write an evaluation method for `Expr` as follows:

```scala
def eval(env: Map[String, Boolean]): Option[Boolean] = this match {
  case Var(s) => env.get(s)
  case a || b => join(a.eval(env), b.eval(env)) by (_ || _)
  case a && b => join(a.eval(env), b.eval(env)) by (_ && _)
  case !(a) => a.eval(env).map(!_)
}
```

(This uses the monadic `join` operation as described on Page 21, adapted for `Option`.)

Such extractors are also automatically generated for case classes, which is the reason they are mainly used for data types that can be pattern matched on. On the other hand, there are types which hide their internal implementation and expose only `apply` and `unapply` methods, for example `Sets`. This allows to introduce an additional layer separating interface from representation. A different usage of them is to provide "additional views" on data. A nice example for this is Scala's regular expression functionality, which uses extractors for retrieving the matched groups of a regex. Consider the following definition of a `Regex` object [OSV08, p. 611]:

```
scala> val decimal = """(-)?(\d+)(\.\d*)?""".r
decimal: scala.util.matching.Regex = (-)?(\d+)(\.\d*)?
```

The `decimal` object now has a variadic `unapply` method, which allows to extract all groups (inside parentheses) to be assigned to valus in a pattern matching context:

```
scala> val decimal(sign, integerpart, decimalpart) = "-1.23"
sign: String = -
integerpart: String = 1
decimalpart: String = .23
```

This can be considered much more readable than using "manual" methods of finding and extracting groups from a match. It also shows a style of using extractors depending not so much on the data matched on, but parametrized by external information.

2.5 Implicit values

Scala has the curious property that it seemingly allows methods to be called on objects on which they are not defined, or parameters to be passed in an automatic, scope dependent way without mentioning them – but all that in a strongly-typed way. This works through the system of "implicits". They are what allows to write `"""(-)?(\d+)(\.\d*)?""".r` without `r` being defined in `java.lang.String`, or to call `List(("b", 1), ("a", 2)).sorted` without having to pass an anonymous `Comparator` for `(String, Int)`.

The unified, "explicit" treatment of such implicit values, allowing *type-directed implicit parameter passing* [OMO10], is a language feature quite unique to Scala. It subsumes in a very powerful way notions existent in other languages, such as type classes, extension methods, monkey-patching, and dynamic scope. Implicits in principle do not really add anything to the core language semantically – they can always be reduced to explicit method calls. However, they allow for some huge reductions of boilerplate code at the call site, for example when converting between representations, extending funtionality, or passing around configuration infomation or context. This feature is ubiquitously used in the standard and other libraries, and several patterns of using implicits have been established.

Implicits come in two flavors: *implicit conversions* and *implicit parameters*. Both in practice often interact with each other, but have different properties when it comes to "discovery" and typing. The foundation of them and their usage are described below in this subsection, in a more theoretic overview. Since some major applications, such as extension methods and type classes, are later in this work explained individually, and other usages are spread throughout many examples, there will be no larger pieces of example code here.

The basic building block of using implicits, both conversions and parameters, are *implicit declarations*. All `val`s, `var`s, `def`s, `object`s and `class`es can be marked as `implicit` (as long as they are not at top-level), and only such marked declarations are subject to implicit resolution. Underlying idea of both is that functions and parameters can be applied without being explicitly named, if they can be uniquely determined from the surrounding *implicit scope*. The exact rules of how this scope is determined are defined in the specification [Ode+14]; Section 6.26 of it describes implicit conversion, and Chapter 7 is dedicated to implicit parameters (plus some features related to them). In the rest of this subsection, these peculiarities are not discussed; they basically amount to searching surrounding imports, companion objects and supertypes, and preferring strictly "closer" and "more specific" declarations to more "distant" ones.

Implicit conversions act on the level of typing context and method names. They are declared by providing an implicit value of a function type, that is, an implicit `def` or `val` with an `apply` method of suitable type. If then an object in a context does not have a compatible type, an implicit conversion is sought and, if available, automatically applied. For example, when we call `qux(baz)`, where `baz: A`, but `qux` does only take parameters of type `B` (and it is not the case that `A <: B`), an implicit conversion function `a2b: A => B` being in scope would be automatically inserted, resulting in an effective call of `qux(a2b(baz))`.

Furthermore, if some method is called on an object, say, `foo.bar(1)`, and the type of `foo` does not contain a method `bar` with that name and signature, then instead of immediately rejecting this as an error, the compiler searches for suitable implicit conversions to types *with that method*. If `foo` has type `A` and the result of `bar` in this context is inferred to be `B`, then a suitable conversion is an implicit value of type `A => { def bar(i: Int): B }` (modulo subtyping relations). When such a conversion, call it `fooWrapper`, can uniquely be found, it is automatically applied; the resulting term behaves the same as `fooWrapper(foo).bar(1)`.

Conversions are subject to an important restriction: they can not be "stacked"; that is, they are not, in a sense, transitive. If `A` is implicitly convertible to `B` via `a2b` in implicit scope, and similar `B` to `C` by `b2c`, there does *not* automatically exist the implicit conversion `b2c(a2b(_))`. This is to simplify the rules and reasoning, but also to prevent exponential search complixity at compilation.

The use of such automatic conversions is manifold: for one, they can simplify code in a setting where datatypes are wrapped in layers, but different layerings essentially mean the same. For example, in the propositional logic DSL from Listing 2.1, `Var("x")` does not contain any more information than just `"x"`, although it is still desirable to represent variables in a dedicated type wrapper. By providing

```
implicit def stringToExpr(s: String ): Var = Var(s)
```

both advantages can be kept: when writing an expression, it is possible to use string literals alone (like `"x" && ~"y"`), but internally, always the case class is used.

A similar use case is the conversion between essentially equal data types not between layers, but between foreign libraries or equally prominent representation. For example, such conversions exist in the Scala collections library under `scala.collection.JavaConversions` for the Java API's equivalents of `Iterable`, `Set`, `Map`, and so on. They are also a very common means to provide Scala-native interfaces to Java libraries using anonymous classes as callbacks; in that way, given

```
implicit def function2ActionListener(f: ActionEvent => Unit) =
  new ActionListener {
    def actionPerformed(event: ActionEvent) = f(event)
  }
```

one does not need to pass a literal `ActionListener` to a Swing event listener, but can directly use an anonymous function, which is much more natural (example taken

from [OSV08, p. 444]). Such conversions are also defined in the `Prelude` module for primitive types and their boxed equivalents (such as `char2Character`).

In a more complex form, implicit conversions to custom types can be used to virtually extend other types, without affecting their actual implementation. This pattern is known under *implicit wrappers* and described in more detail in Subsection 3.1. It is quite regularly made use of in the standard library to extend the (sometimes rather poor, or at least unconvenient for Scala) interface of Java standard classes for all array types, primitive types, and strings, which are, for example, augmented with higher-order collection methods.

Implicit parameters, in contrast to conversion, act by the automatic insertion of method parameters. Parameters subject to implicit resolution must be declared in their own parameter group (see Page 17 on currying). There can be only one such group, which must come last in the series of parameter lists. Take the following method, assumed to be defined in class `X`:

```
def fn(a: A)(implicit b: B): X
```

Given values `x`, `a`, and `b`, `fn` can always be called in a regular fashion: `x.fn(a)(b)`. However, when a value of type `B` is in implicit scope, say `implB` (declared, e.g., as an `implicit val`), we are allowed to leave out the second parameter group, and just write `x.fn(a)` – which automatically will again resolve in `x.fn(a)(implB)`. (It should be noted that it is always possible to explicitly pass values into implicit argument positions, as they are just syntactic sugar – so no generality is lost, if "overrides" are desired.)

This kind of parameter passing is mostly used when an interface requires to pass a lot of callbacks or context objects. By making these explicit in the type, but implicit for the call site, the boilerplate effort for the user of the interface can be reduced, as well as the readablility improved, while no "interface information" gets lost. The classic example of this is the above-mentioned need to pass a `Comparer` to Java's `Collections.sort`, when trying to sort something which is not a base type. When the sorting method would take the `Comparer` as an implicit argument, we could provide it implicitly once and for all for every new type, and never see it being passed explicitly (except cases when we want to change sort order in special ways, which are however better fitted for `sortBy`). In fact, Scala's standard library's `sorted` works exactly that way, and takes an implicit `Ordering[T]` for the type `T` to be sorted. In Subsection 3.2, the generalization of this powerful pattern is explained in more detail.

A non-type-class example of implicit parameters is `scala.concurrent.Future`, most of which's combinators take an implicit `ExecutionContext`, like the method

```
foreach[U](f: T => U)(implicit executor: ExecutionContext): Unit
```

That way, the kind of asyncronous execution needed for executing concrete `Future` callbacks is always available internally, but never has to explicitly passed around. It can be changed globally by importing or defining a new implicit `ExecutionContext` in close enough scope.

Finally, a word of caution: using implicits in a careless way *can* lead to very hard to find sources of error. This happens especially if an implicit conversion unrealizedly is in scope, applies to some value, and then fails. The author learned this the "hard way" when an expression of the form `foo + 12`, which was mysteriously typed as `Int` in the IDE, failed at runtime with a `NoSuchElementException` – only after inspecting the bytecode, it could be found out that there was an implicit `Map` from `foo`'s type to `Int` lying around, whose `apply` method was treated as a conversion and came into effect on `foo` (which the `Map` did not contain); even worse, the implicit value was even "leaked" through the external interface. The problem was then resolved by wrapping the `Map` in a purely internal case class, and changing the type of the implicit parameters for which it was originally thought accordingly.

To avoid such situations, two general advices concerning implicits should in almost any situation be followed:

- Implicit conversion should never fail at runtime with exceptions, or otherwise.
- Implicit parameters, result types of implicit conversions, and every implicit declaration, should have as specific and customized types as possible – if in doubt, they should be wrapped in a class just for that purpose.

In summary: let the compiler do the work for you – but always be in control of it…

2.6 Macros

In contrast to the other subsections, this one tries to give more of an overview and background of its topics than examples and concrete use cases. This is because macros are relatively new, and will probably change soon. Nevertheless, they are an important (and practically used!) "last resort" for certain kinds of problems in the scope of this work, and certainly deserve to be mentioned in an overview of DSL techniques.

Macros are a traditional topic when it comes to DSLs, though their prominence in software development has declined over time. They are in their many forms a much more general way to extend the expressiveness of a language, not only by adding seemingly impossible syntax, but especially by providing semantic extensions via changing or introducing non-standard evaluation behaviour, which they can do because they operate *before* function evalulation. Macros in the sense used here (as proper syntactic transformations or *metaprograms*, not only text replacement) were originally introduced in LISP, where they come out in a natural way from the fact that S-expressions (defining functions) are able to be applied to other S-expressions (representing functions) – so they are, in a way, a lucky side effekt of the syntactic simplicity of LISP [McC60].

As indicated, "proper" macros are rare in today's programming languages. Frequently, especially in object-oriented languages, we find so called *reflection* facilities in the standard libraries, which allow to inspect and sometimes manipulate the representation of programs at runtime. Or, there are APIs for the compiler or inter-

preter allowing to extend them. Both of this has been existing in Scala practically for free since a long time, because of the already available Java reflection API [JavaAPI, `java.lang.reflect`]. The reasons to avoid macros are, for one, that in most languages different from LISP complex reification and representation support is needed, which is more complicated to handle than to just write code differently; additionally, macros can be considered problematic from the point of view of certain software development philosophies or styles, since they are hard to test and somewhat opaque. As an interesting example of a language which does have a very structured approach to metaprogramming in the sense used here, though, we could mention the statistics language R [R15].[11]

Still, there recently has been put effort into implementing a proper macro system for Scala [BO13], along with a quoting syntax [SBO13], which significantly reduces difficulty and improves readability of macro implementations. These implementations are available in newer Scala versions (since 2.10), and, although considered experimental, are already widely used in libraries. There is currently a project called `scala.meta`[12] which aims at improving and unifying all metaprogramming facilities available in Scala (including reflection) and will provide a completely new API in the future.

THE DESIRE TO WRITE FUNCTIONS that transform not their input values, but the whole expression trees they are called on, is not as exotic as it may seem at first. In fact, the well-known C preprocessor does exactly that, though in a very limited form – which is the reason the "functions" that can be `#defined` are called macros, too.

People write macros at this level for two reasons: inlining, and non-evaluation. The first one is quite discussable – nowadays, the overuse of preprocessor macros for "optimization" is often frowned upon, as they can lead to strange errors when not thoughtfully crafted, and explicit inlining is mostly considered useless with modern compilers, since their optimizations turn out to be superiour to anything that can be influenced manually. In Scala, for this purpose, there is an annotation `@inline` (and a corresponding `@noinline`) in the standard library, serving as hints for compilers. Additionally, there have been put efforts into the improvement of the performance of implicit wrappers, which have lead to *value classes* [Ode+12] (they allow to discard a wrapper class at runtime, similar to Haskell's `newtypes`).

The second traditional reason to use macros, non-evaluation, has its classic use case in the writing of debug functions and other condition-like code simulating statements, which cannot easily be written in the form of functions, since parameter passing in conventional imperative semantics is strict and always evaluates the arguments. However, as we have seen, this is easily possible in Scala, by passing blocks, evaluated on demand, as call-by-name parameters, which is a very common technique present everywhere in the libraries.

[11] Specifically concerning R's metaprogramming capabilities, see `https://cran.r-project.org/doc/manuals/r-release/R-lang.html#Computing-on-the-language` (visited on 2015-11-14).

[12] `http://scalameta.org/` (visited on 2015-04-26).

But going a bit further with the debug example, we soon arrive at the limits of regular syntax and semantics. For example, it is impossible with non-macro functions to implement an assertion function whose error message automatically contains the actual condition it was given; that is the reason `assert` from the standard library takes a string as its second parameter. Obviously, it would be desirable if the expression `assert(foo > 0)` would, on error, automatically print a message like `"Assertion 'foo > 0' was violated"`, extracting the text `foo > 0` from its argument. (This is certainly possible with preprocessor macros, but unsafe, since their expansion happens before compilation on the raw program text.)

With macros, we have the ability not to deal with values, but with expressions, and this problem can be solved: we could take the given expression `foo > 0`, convert it to a string representation, and return an expression containing a non-macro assertion with the `foo > 0` as condition and a message built from the string representation:

```
macroAssert(foo > 0) ⇝ assert(foo > 0, "Expected foo > 0!")
```

As soon as we have macros at our disposal, a lot more power than in this simple example becomes available. In current Scala libraries using macros, they are typically used for two purposes: to rearrange or check closures/blocks, and to automatically generate boilerplate implementations. Both cases are normally used in DSLs (or even "meta-DSLs", simplifying the implementation of libraries).

The first of these use cases mostly involves transforming a closure into something more complicated, which nevertheless can be recovered from a simple block of code. One typical example of this would be the `async` syntax [HZ13], which provides a DSL for computations with futures.[13] Remember that we can combine future values in various ways by using their monadic interface, which gives the ability to use for comprehensions:

```
def slowCalcFuture: Future[Int] = ...
val future1 = slowCalcFuture
val future2 = slowCalcFuture

def combined: Future[Int] = for {
  r1 <- future1
  r2 <- future2
} yield r1 + r2
```

The library simply provides a macro `async`, and a marker method `await`, which allow the above comprehension to be written like this:

```
def combined: Future[Int] = async {
  await(slowCalcFuture) + await(slowCalcFuture)
}
```

[13] Code available at `https://github.com/scala/async` (visited on 2015-05-17). The subsequent examples are also taken from the documentation provided there.

This macro, instead of transforming the code to monad function calls, like it would be done with a comprehension, builds an object containing a state machine which is registered at the futures' `onComplete` handlers. This is also more efficient, since it does not involve multiple anonymous classes for the intermediate closures.

The async/await pattern is also implemented in C# (since version 5.0), and is becoming popular in other languages, too (e.g., Python [Sel15]). However, using macros, it can be implemented in Scala purely as a library – the underlying transformations, as applied by the C# compiler and the `async` macro, are quite the same [Tor10].

In a similar direction goes the `spores` proposal [MOH13]. Its purpose is to allow an interface taking a closure to be guaranteed that no local references (like `this`) are captured in a dangerous way. Such a guarantee is necessary for tasks like serialization of functions, or passing around closures in distributed systems. This is done by wrapping a function literal in a new type `Spore[-T, +R]`, but allowing them only to be constructed through a macro which enforces that all captured variables are explicitly copied before usage (example taken from the cited SIP):

```
val s = spore {
  val h = helper  // explicitly copy this reference
  (x: Int) => {
    val result = x + " " + h.toString
    println("The result is: " + result)
  }
}
```

Using any variables not defined in the header in the lambda expression will result in a compilation error.

Many examples for the second popular macro usage, automatic implementation of "redundant" code, can be found in the `shapeless` library.[14] This library provides, among other things, a wide range of higher-rank polymorphic combinators; that is, typesafe functions working on general products and coproducts, like folds over arbitrary tuples. These things are mostly achieved by using powerful type classes (Subsection 3.2), which do not, however, have to be provided by the user. Instead, `shapeless` relies on *automatic type class derivation*. This method is based on *implicit macros* – which are just implicit instances, for which the code is automatically generated by macros at compile time. These derivations usually just involve inspecting the code of case classes and producing type class witness `objects` according to their shape and type.

[14] `https://github.com/milessabin/shapeless` (visited on 2015-05-17).

3 Patterns for DSL Implementation

This section is based on the previous one, and shows examples of how the Scala features introduced there (and some more) can be combined into patterns typically used in DSLs. By "patterns", not the strict architectural layouts, as used in object oriented design, are meant. Rather, this is a collection of ways commonly used to achieve a certain goal: implementing DSLs in a syntactically and functionally satisfying way. It is noteworthy that almost all of the patterns are virtually native to Scala, in the sense that the features used were introduced for the very purpose of simplifying their application: traits for mixins, implicits for type classes and extension wrappers, the syntactic flexibility of method calls for combinators and other kinds of syntactic DSL helpers.

Additionally, as showcases of these patterns and the interplay of various Scala techniques, a number of existing DSL libraries will be mentioned or used as examples.

3.1 Extensions and Rich Wrappers

One of the simplest forms of DSLs is to extend existing data types with new operations that are frequently needed for the domain under consideration, to provide shorter functions to interoperate with new types, or to combine values into domain constructs using more concise syntax.

One common example of this would be `Range`, as it exists in the Scala prelude. A `Range` object represents a contiguous sequence of integers, which can be stored efficiently only by its first and last value (and optionally, step size). The conventional way to construct such a value would probably be through a factory object like `Range(1, 10)`, or, if we want to avoid ambiguities, `Range.inclusive(1, 10)`. This is, however, not as easy as it could be – there are ways of providing special syntax for ranges, which can be more readable. For example, in Haskell, one could write `[1..10]` for the same thing.

Fortunately, Scala has implicits and a very practical method calling syntax. Using both, we can write code like in Listing 3.1. There, we simply create an implicit wrapper class, closing over the first `Int` and giving it methods `to` and `until`, which then construct the respective ranges with a second `Int`. Using such a class, we now

```
// suppose {in|ex}clusiveRange: (Int, Int) => Range
implicit class IntRangeOps(self: Int) {
  def to(bound: Int): Range = inclusiveRange(self, bound)
  def until(bound: Int): Range = exclusiveRange(self, bound)
}
```

Listing 3.1: Implicit wrapper class to build ranges from integers, for a given type `Range` with a construction function.

```
// suppose definition: class Value[D <: Dimension]
implicit class dimensionSymbols(d: Double) {
  def meters = new Value[Meter](d)
  def seconds = new Value[Second](d)
  def amperes = new Value[Ampere](d)
  def coloumbs = new Value[Ampere * Second](d)
}
```

Listing 3.2: Implicit wrapper to construct `Values`, which are type-safely dimensioned doubles. The primitive units are definded literally and can be applied in postfix notation; `of` allows to select an arbitrary dimension as its type argument (such as `9.8.of[Meter / Second / Second]`). Some units have been left out here.

have new syntax like `1 to 10`.[15] Implicit classes are just syntactic sugar of a class declaration with an additional implicit `def` for the respective conversion; they were introduced specifially for this use case. A lot of such functionality-patching extensions can be found in the Scala standard library. Besides `to` and `until`, there are also `StringOps` and `WrappedString`, and similarly `ArrayOps` and `WrappedArray`.

The code in Listing 3.2[16] contains a further example of how implicit wrappers are often utilized: to provide a nicer construction experience than regular constructors. The implicits listed there allow to write numbers in SI units in a more natural way, e.g., `42.0 meters`, instead of `new Value[Meter](42.0)`. Since there are more possibilities of units than just the bases, two means of including them are shown: one could, for convenience, provide similar constructors for all well-known combinations, like it is done in `coloumbs`; or, a general interface like `of` could be provided, allowing to construct arbitrarily dimensioned values by a type argument. A similar system (although not using type parameters) can also be found in the standard library at `scala.concurrent.Duration` for time durations, which get mostly used for specifying timeouts in concurrent environments.

There is one caveat when using many implicit wrapper classes: although all calls look like they would work directly on the objects, they are often in fact passed through multiple layers of nested calls, which can lead to performance loss. Such cases should of course be inspected with a profiler before applying premature optimization; still, if necessarry, there are some tricks available, which were already brought up in Subsection 2.6. For one, there is the `@inline` annotation, which can

[15] Note that, when trying this out in an interpreter in the shown form, there will be a conflict with the implicit conversions of the same names defined in the prelude. A solution is to simply rename the methods.

[16] In this case, the accompanying example on Github is far more experimental than the others, as it contains an *unfinished* system trying to proove equality of dimensions at type level.

```scala
trait Read[T] {
  def read(s: String): T
}

object Read {
  implicit object stringIsRead extends Read[String] {
    def read(s: String) = s
  }

  implicit object boolIsRead extends Read[Boolean] {
    def read(s: String) = s match {
      case "true" => true
      case "false" => false
    }
  }
}
```

Listing 3.3: Read type class, together with implementations for strings and booleans.

be used to hint the compiler to optimize implicit conversion calls. Then, in newer Scala versions, there are *value classes* available [Ode+12]. These allow, if certain conditions are fulfilled, to erase wrapper types at runtime, and to replace implicit calls by static applications. Furthermore, if this is not enough, macros can be used to specialize implicit wrappers again, and get rid of some layers.

3.2 Type Classes

Type classes are another pattern built on implicits. They are a language construct originally introduced by Haskell, and comparable to interfaces in object orientation, in that they allow to specify operations that a type must support. The main purpose of them is to constrain generic functions and to write implementations against an interface; however, they are in a few ways more general than conventional interfaces, and solve some peculiarities of them. Among type theorists, type classes are said to unify advantages of both *ad-hoc polymorphism* (overloading) and *parametric polymorphism* (type parametric functions); this is described in [WB89], while the terminology goes back to [Stroo]. A detailed description of how Haskell's type class concept has been translated into Scala, including an overview of their applications and generalizations using implicits, can be found in [OMO10].

To illustrate how type classes work, and how they are different from interface inheritance (embodied in Scala by "normal" trait inheritance), consider Listing 3.3. This example shows a `Read` type class – a concept from Haskell, which says "we can parse a value of a type from a string". This possibility is represented by the parametrized trait `Read`. The crucial difference from conventional interfaces is that some type `T`, implementing the type class, is not supposed to inherit from `Read` – instead, a *witness object* of type `Read[T]` should be provided. The witness object, in this case encapsulating only one function, can then be used by third parties to perform the desired operation. In the example, such objects are shown for the trivial `String` instance, and for `Boolean`. When another method uses the type class to

```
implicit class StringReadOps(val self: String)
{
  def readAs[T](implicit readT: Read[T]): T = readT.read(self)
}
```

Listing 3.4: Wrapper object for the Read type class.

constrain a parameter, the witness needs to be passed in as an additional parameter, which can then be used inside the method to operate on the constrained type.

This sounds like a syntactic hassle at the call site, similar to Java's handling of, for example, custom equality. And in fact, passing an anonymous `Comparer<T>` to a sorting method is just an explicit variant of the concept of a type class. However, in both Haskell and Scala, type classes are empowered to prevent such reduncancy: in both languages, the witness dictionary is passed implicitly to functions requiring a parameter to be instance of a type class. In Haskell, this passing is automatically resolved and inserted by the compiler [HB90], while in Scala, it is made explicit at the definition side by using `implicit` parameters. Consider the following typical method definition:

```
def readAll[T](l: List[String])(implicit readT: Read[T]): List[T] =
  l map readT.read
```

This reads every element of a list into a specified type, if the elements implement `Read`. Calling this function is simple, provided that the necessary implicit value is in scope:

```
scala> readAll[Boolean](List("true", "false", "false"))
res0: List[Boolean] = List(true, false, false)
```

Of course, the result type always has to be provided explicitly, since it cannot be inferred from a string. Similarly, one type class implementation can also depend on another one; for instance, all `Numeric` instances can be `read`, at least from integers:

```
implicit def numericIsRead[N](implicit numN: Numeric[N]): Read[N] =
  new Read[N] {
    def read(s: String) = numN.fromInt(s.toInt)
  }
```

In this form, method calling looks seamless and unsuspicious, but there is still an overhead involved every time the operations are actually used on values. For this purpose, type classes often provide wrapper objects (see Subsection 3.1) with operations for objects that can be used as first argument of a method of a type class. The names of these often end with `*Ops`. For the `Read` example, such a wrapper object is shown in Listing 3.4. In this form, the explicit mentioning of `implicit` parameters is constrained to a minimum, and type classes can be used as if they just provided the right methods on the right types – all without inheritance:

```
def readAll[T: Read](l: List[String]): List[T] = l map _.readAs[T]
```

This example uses one more feature to fully eliminate mentioning the implicit passing: `T` is constrained by a *context bound*, since we do not need the witness object itself

anymore. Context bounds are automatically converted by the compiler to implicit parameters, and come very close to the type class syntax of Haskell. Still, they require more effort at the side of the library, in that for seamless usage, an `Ops`-wrapper has to be written (otherwise, the user of the type class can always obtain the witness explicitly by the prelude function `implicitly[T]` [Ode+14, Chapter 12.5]).

Furthermore, there is one more thing type classes allow: They can be "stacked", which means using a type class constraint inside the definition of another instance. For example, when we want provide a `Read` instance for maps, a suitable definition would have the following signature:

```
implicit def mapIsRead[K, V](implicit readK: Read[K],
                                      readV: Read[V]): Read[Map[K, V]]
```

The implicit witness for the wrapping type requires itself a witness for the contained key and value types. Note that this "stacking" of implicits does not fall under the rules of implicit scope resolution for values, which allow only one level of implicit wrapping – chains of implicit parameter requirements are followed arbitrarily deep.

The type class pattern, as over-complicated as it may sound on first sight, is not only a syntactic trick. It has the following actual semantic advantages over trait inheritance when it comes to code architecture:

- It does not only allow separating interface from implementation, but also separation of *data implementation* (of the class that implements the interface) and *functionality implementation* – most additional functions and combinators provided by the type class can have their own, third place. This leads to a second property:
- Type classes allow to extend types independently of their declaration. That means, a type class implementation can always be added at a later time, and somewhere different from the original class. In that way, the provide a kind of "interface for extension methods".

But type classes are also strictly more expressive than conventional subtyping polymorphism [WB89]. Consider a hypothetical trait for a semigroup:

```
trait Semigroup {
  def plus(other: Semigroup): Semigroup
}
```

First, with this idea, every type being a semigroup would have to be declared as such at its definition, by inheriting from the trait – there would be no way of "saying that later". But more importantly, this code does not even correctly capture the functionality we want to have, since it operates only on the `Semigroup` base type – and by subtyping, the respective parameters can also be in every other semigroup. The operation could not even be implemented properly in an inheriting class, since the parameter type is too general to access specific members of `this`. This problem is known (remember the peculiarities of defining an `equals` method!), and has lead to a solution using generics (or templates), known under the name "curiously recurring

template pattern" [Cop95], in which a type inherits from an interface with itself as a parameter (like `AddableInt extends Semigroup<AddableInt>`). This still is somehow tedious and does not resolve all issues; in particular, it still cannot express a type class like the following:

```
trait Monad[M[_]] {
  def pure[A](a: A): M[A]
  def join[A](mma: M[M[A]]): M[A]
}
```

This is because neither of `Monad`'s methods are properly expressible as methods of some `Monad` object – `pure` is a factory, which should probably be a static method; and `join` can only operate on objects of *nested* monad type, requiring a form of constraining `this`, which is not possible to express in Scala (or the JVM in general).

From this we see that, while traits only allow to specify methods of the form `T => X1 => ··· => Xn`, where `T` is the base type, type classes can specify an interface with functions where the constrained type occurs multiple times and at any position.

Examples of type-class based libraries "in the wild" are Spire[17], which is a numerics library using type classes to form a mathematical hierarchy of algebraic constructs (such as semigroup → monoid → group → semiring), and to abstract out kinds of "number types", like integral numbers, rationals, or complex numbers. In a similar direction goes Algebird[18], which utilizes mostly monoids and semigroups to implement various algorithms with generalized matrices on map-reduce style platforms.

3.3 Combinator Interfaces

The extensive use of combinators (infix functions, mostly symbolic ones, to shorten expressions) for DSLs has probably been popularized by Haskell. The reason for exactly this language can be found in the theoretical background of its inventors and users, which for the most part are strongly influenced by mathematics and tend to apply similar symbolic conciseness to programming. But the "philosophy" behind combinator interfaces is not (only) to replace function names and method calls by (often cryptic) symbols; rather, they try to factor out certain kinds of patterns and provide a succinct way of writing them. Such patterns are those for which traditional inline notations exist. Most prominently, this is the mathematical notation for calculations containing things like vectors or matrices; however, there are other patterns, such as pipe-and-filter (known from UNIX shells), or grammar notations, like regular expressions or the Backus-Naur-Form.

Starting from these patterns, and equipped with a frictionless treatment of operators as functions, there has since long been established a common "basic set" of operators in the Haskell standard library – not only for DSLs. These have layed

[17] `https://github.com/non/spire` (visited on 2015-06-03).
[18] `https://github.com/twitter/algebird` (visited on 2015-06-03).

```
-- Increments all of the major versions of an array of JSON objects.
someString ^.. _JSON        -- a parser/printer prism
. _Array       -- another prism
. traverse     -- a traversal (using Data.Traversable on Aeson's Vector)
. _Object      -- yet another prism
. ix "version" -- a traversal across a "map-like thing"
. _1           -- a lens into a tuple (major, minor, patch)
%~ succ        -- apply a function to our deeply focused lens
```

Listing 3.5: Usage example of lenses, a popular pattern for purely functional structural traversal/modification in Haskell, which utilizes an extreme variety of combinators. The `(.)` operator is nothing more than regular function composition; using it, various lenses are combined to "focus" on the part of interest, on which then the function succ is applied.[19]

the basic and inspiration for many other libraries using combinators. A very good example of extremely rich Haskell combinators is the Lens package,[20] which provides combinators for generalized getters, setters and modifiers for nested data structures. In Listing 3.5, an example of its usage is shown: the code there describes a way to modify the major version number of a given JSON object.

In Scala, the language often does not allow (or at least makes very impractical) such an extreme use of combinators as in Haskell (this is due to the restrictions of how precedence of operators is determined (cf. Page 11), and to the lack of automatic currying). Nevertheless, there are places and applications where they turn out to be useful and are commonly employed. One such application is parser combinators. These have been popularized first by Haskell's efficiently implemented Parsec library [LM01][21], but have since been ported to various other languages – including Scala, where they are available as a separate module `scala-parser-combinators`[22].

In Listing 3.6, an example of its usage is shown for a simple grammar for LISP-style S-expressions (which look like `(foo (cons 0 '(1 2 3)) (bar ()))`). Like in a BNF grammar, nonterminals are given their own names; they are implemented as individual `Parser` objects mutually calling each other. These nonterminal-parsers are then combined with each other and primitive parsers for terminals, using a range of operators, many of which have mnemonic names; for example, the expression

```
whiteSpace.? ~> (readMacro | cons | atom)
```

means "parse optional whitespace, discard it, then apply whichever of `readMacro`, `cons` and `atom` succeeds, and return its result" – this uses a mixture of BNF alternation with `|`, the regex "optional" modifier `?`, and Haskell's left-discarding applicative combinator, which originally is `*>`, but has been changed to `~>` in Scala, where it is also congruent with the sequencing operator `~`.

[19] Example taken from `https://www.fpcomplete.com/user/tel/a-little-lens-starter-tutorial` (visited on 2015-06-04).

[20] `http://hackage.haskell.org/package/lens` (visited on 2015-06-04). A very good point of overview is `Control.Lens.Lens`, which already contains exotically-looking combinators like `(<<%@~)`.

[21] `http://hackage.haskell.org/package/parsec` (visited on 2015-06-04).

[22] `http://www.scala-lang.org/files/archive/api/2.11.x/scala-parser-combinators/#package` (visited on 2015-06-05).

```
object SExpParser extends RegexParsers {
  override val skipWhitespace = false

  def sexpr: Parser[SExpr] = whiteSpace.? ~> (readMacro | cons | atom)

  def readMacro: Parser[SExpr] =
    (readMacroIdentifier ~ sexpr) <~ whiteSpace.? ^^ {
      case s~e => ReadMacro(s, e)
    }
  def cons: Parser[SExpr] =
    parenthesized(rep(sexpr) ^^ ConsList) <~ whiteSpace.?
  def atom: Parser[SExpr] = (identifier ^^ Atom) <~ whiteSpace.?

  def parenthesized[T](p: Parser[T]) =
    "(" ~ whiteSpace.? ~> p <~ whiteSpace.? ~ ")"
  def identifier: Parser[String] = "[^()' ]+".r
  def readMacroIdentifier: Parser[String] = Quote.macroCharacter
}
```

Listing 3.6: Parser for a simple S-expression syntax for LISP, including quotes. The `^^` combinator is essentially (f)map – it lifts a function to transform the result of a successful parser.

In contrast to traditional BNF, it is also possible to define "parametrized symbols" in the form of functions (such as `parenthesized`). This is an important point. Contrary to specialized, external languages (in this case, parser generators such as ANTLR), embedded combinator libraries have the advantage of and should be designed to being able to interact with the surrounding language *inline*. This is not only done by using language constructs to form new parsers (parser-transformers like `parenthesized`), but also by being able to embed Scala in parsers with function combinators like `^^`. In that way, parser combinators are an excellent example of SICP's principles of metalinguistic abstraction: they neatly combine means of abstraction, combination, and factoring out patterns, by seamlessly interacting with the functionalities of the host language.

When designing a combinator interface in Scala, not only the conciseness of expressions should be respected, but also the conventions and habits of the users of the language. For example, while in Haskell the `atom` parser might be expressed as

```
Atom <$> identifier <* optional whitespace
```

this is rather impractical in Scala, since the higher-order application would not work in expected ways. Instead, using `^^` with an anonymous function is the preferred way to map over a parser. On the other hand, the usage of methods for changing parsers, like in `whitespace.?`, is not easily possible in Haskell, but quite effortless in Scala, and forms thus a good way of modifying combinators. The same would also work in `cons` (line 10), where `rep(sexpr)` could also be written as `sexpr.+`; but the tradeoff between symbolic names and static functions in a case like this is more one of personal style. In general, a good solution would probably take into account both backgrounds and provide method-style as well as combinator-style parts of the interface, allowing the user to choose a suitable mixture.

3.4 Objects and Modules, Mixins and Cake

This subsection's purpose is to introduce a few practical notions of modularization and organization of code parts. They are the most important concepts for this used in Scala, and shown here in order to demonstrate how a good "user experience" for a library can be constructed; besides, also some specific tricks using traits are shown, which are examles of how to provide modularized and customizable behaviour for classes.

As it was stated in the introduction, Scala has a highly sophisticated module concept – in this respect, it allows much more flexibility than many other languages. Still, the underlying building block of this system is just one thing: the `object` construct. An `object` is basically just a syntactic sugar for declaring a singleton instance of some class at top level, with the convenience of having the same syntax as a class. However, `object` declarations have really more power than plain instances; most importantly

- every `object` forms a *module* – a separate namespace, that can be `imported` otherwhere, but can also contain its own state; furthermore,
- `objects` can be used as *companion objects* for classes or traits, in which case their members generalize the notion of static members known from other languages; and finally,
- *package objects* can be used to provide free functions and objects at the top level of a package, which are automatically available at import.

In the following, the term "object" will mostly denote such singletons declared by `object`, not just arbitrary instances.

In the first, simple variant, objects can be used to modularize small pieces of code, without having to use the package system. This is useful if one wants to provide helper methods or implicits to be used for a specific DSL, and provide them in an extra namespace. It can also be used to bundle features in a kind of "overlapping" module – Scala has no explicit notion of exporting from a package (everything that is not marked as package private, is considered public), so sometimes it is practical to collect a range of members in an object and relay their definitions to other namespaces, in that way providing a combined reexport.

A further common pattern for this use case is to separate code using traits. Since traits, unlike interfaces, are not necessarily something purely abstract, but can be fully concrete, they are often used to just split parts of an implementation (logically belonging together) into multiple internal parts, and mix them together into one final exported object – which in the most extreme case does not even need to contain any other code. A perfect example of this is the organization of the Scalaz library.[23] Scalaz is a very rich package, but tries to allow very granular import behaviour; every part of its functionality can be imported separately (e.g., `import scalaz.Id._`).

[23] `https://github.com/scalaz/scalaz` (visited on 2015-05-16).

```
object Scalaz
  extends StateFunctions
  with syntax.ToTypeClassOps
  with syntax.ToDataOps
  with std.AllInstances
  with std.AllFunctions
  with syntax.std.ToAllStdOps
  with IdInstances
```

Listing 3.7: The definition of the `Scalaz` object (defined in file `scalaz/core/src/main/scala/scalaz/Scalaz.scala`), collecting functionality from a range of implementation traits.

However, if one wants to bring into scope the full range of features, one only needs to import two things:

```
import scalaz._
import Scalaz._
```

The first one is the package itself, which contains mostly type synonyms and some very basic instances; the second line, importing `Scalaz._`, imports the members of the object listed in Listing 3.7: this object has no own members, but accumulates the members of seven other traits providing all kinds of general functionality contained in the library, from syntactic helpers to type class instances for standard library types.

If the set of helpers is specific to a certain class or trait, it can be put in the *companion object* of that class. Such an object must have the same name and be defined in the same file as the companion class. Defining members in the companion has the advantage that they then have access to the private members of the class; in that way, companions incorporate everything that would be `static` in Java. Additionally, the companion object is taken into account for implicit scope resolution, so it makes sense to define known type class instances there.

A common pattern for implementing factories in Scala is to use the companion object for construction and destruction; that is, to define `apply` and `unapply` there, and hide the actual (maybe more complicated) implementation from the user. This has the additional benefit that the implementation can later be swapped without interface changes. An example for such an implementation is shown in Listing 3.8; this uses a rather unusable underlying implementation of lists, which nevertheless can be used in a normal way via the companion object:

```
scala> ChurchList(1, 2) match { case ChurchList(x, xs) => s"Full:
  ↳ Cons($x, $xs)"; case Empty => "Empty!" }
res0: String = Full: Cons(1, 2)

scala> ChurchList() match { case ChurchList(x, xs) => s"Full: Cons($x,
  ↳ $xs)"; case Empty => "Empty!" }
res1: String = Empty!
```

The definition of its behaviour is part of the sealed trait `ChurchList`, but the contents

```scala
sealed trait ChurchList[+T] {
  def fold[K]: K => (T => K => K) => K
}

object ChurchList {
  def apply[T](l: T*): ChurchList[T] = l.toList match {
    case Nil => Empty
    case x::xs => new ChurchList[T] {
      val rest = ChurchList[T](xs: _*)
      def fold[K]: K => (T => K => K) => K =
        nil => plus => plus(x)(rest.fold(nil)(plus))
    }
  }

  def unapplySeq[T](l: ChurchList[T]): Option[Seq[T]] =
    Some(l.fold[List[T]](Nil)(x => xs => x::xs))
}

object Empty extends ChurchList[Nothing] {
  def fold[K] = nil => _ => nil
}
```

Listing 3.8: An implementation for polymorphic Church encoded lists. The `apply` and `unapply` methods are the only way to construct and deconstruct them; from a more theoretical point of view, they form the isomorphism between `ChurchList[T]` and the builtin `List[T]`. `Empty` is defined as a dedicated object to allow sharing and to make pattern matching look more natural.

can only be accessed through the static methods from the companion (`sealed` prevents implementing the trait from elsewhere than the defining file, thus preventing "direct" creation without the factory methods).

The third form of modularization throught objects are package objects. They are usually defined in a separate file in the package's directory, called `package.scala`, and differ in their definition only by the additional keyword `package`; as an example, consider again an according (greatly simplified) definition in Scalaz[24]:

```scala
package object scalaz {
  import Id._
  implicit val idInstance: Traverse[Id] with ... with Cozip[Id] = Id.id
  type Tagged[T] = { type Tag = T }
  type @@[T, Tag] = T with Tagged[Tag]
  ⋮
}
```

The contents of package objects are always imported when the respective package name is imported, and provide top-level "free definitions". Usual contents of them are type synonyms, which are desired to be avaliable globally (since `type` declarations are only allowed within classes or objects), and widely used implicits for the package. Often, they also contain various small helpers such as wrappers or functions intended

[24] `https://github.com/scalaz/scalaz/blob/series/7.2.x/core/src/main/scala/scalaz/package.scala`, simplified (visited on 2015-11-26).

to be used frequently, which are not considered complex or important enough to deserve their own files or modules.

Besides these patterns using objects, there are also a variety of useful things that can be done with traits, other than using them simply as interfaces. One of the common usages has already been mentioned, namely, the splitting of code among several implementation traits. A variant of this is providing helper methods and implicits to be used with a class or type class in an own trait, and mix that into the class itself or the package object. That way, all utility functions using some type are always imported together with it (these are often found in combination with type classes' `Ops`-wrappers mentioned on Page 37).

Other patterns using traits concern methods of modularizing optional behaviour of classes. This is often wished when there is one "main" class, providing a basic behaviour or functionality by inheritance, and multiple additional features for it, which are however optional. The conventional way to solve such a scenario would be an OOP pattern like Decorator; however, all these are subsumed under traits.

There are two ways to conveniently provide "decoration traits", both of which rely on a stronger coupling between the trait and the basic class that mere implementation traits. One variant is to give a trait a *self-type annotation*:

```
class A {
  def modifyString(s: String) = s * 2
}

trait T { this: A =>
  def modifyAllStrings(ss: Seq[String]) = ss map (modifyString(_))
}
```

Such an annotation consists of the statement `this: A =>` at the beginning of the body, which means that "the type of `this` must be `A` (or a subtype thereof)". In this way, `T` can refer to `A`'s public methods as its own, since it "knows" that the resulting actual type will always be an `A`:

```
scala> (new A with T).modifyAllStrings(List("sdf", "ab", ""))
res0: Seq[String] = List(sdfsdf, abab, "")
```

This kind of mixin is useful when providing additional features to a class, such as testing methods, or special optional syntax. Combining this with leaving members abstract in the "main" class, this pattern is frequently used for dependency injection and known as *cake pattern*, since it constructs a final implementation out of a variety of available layers (such as logging, testing, persistence, …).

Traits with self-type annotations however cannot influence the basic behaviour that was "already there". For that purpose, a more sophisticated kind of mixins can be employed: this second variant is a bit more interesting from an object-oriented perspective. It exploits the fact that Scala trait inheritance uses *dynamic super-type resolution*. That is, when mixing traits from an inheritance hierarchy, behaviour can not only be stacked, but overridden in a well-defined manner. Consider the following

trait, continuing the example from above:

```
trait Logging extends A {
  override def modifyString(s: String) = {
    println(s"[Log] Old string was: $s")
    super.modifyString(s)
  }
}
```

If we now say

```
scala> (new A with Logging).modifyString("sdf")
[Log] Old string was: sdf
res0: String = sdfsdf
```

we see that the old behaviour has in fact be extended with the trait. This can be used to combine multiple available behaviours in a "vertical" way, instead of only "horizontally", as with the first variant – the different implementations are added above each other. The dynamic mixin inheritance provided by Scala is a practical and powerful alternative to other styles of multiple inheritance. It differs from most other approaches by the fact that calls to `super` are not resolved at the place where they appear – rather, in the final object, all mixed-in traits are *linearized* according to a specified order, which is then used to resolve the call [OSV08, Chapter 12.6].

The relation of all these patterns to DSLs is the fact that they can be used to provide a much more natural interface for DSLs which are thought to be implemented as classes. A common way to design a library for this is to have an abstract class containing the basic functionality, and intend it to be extended on a case-by-case basis. So, for one specific instance of the usage of the functionality, the user would create a new subclass or even subobject, and implement the concrete behaviour in the body of the class (either in the primary constructor, or by implementing abstract members). Then, additional functionality can be provided via mixins; this can range from mixing in the actual DSL syntax to just adding some debug functionality.

A nice example of this kind of interface is shown by the Akka library,[25] which superseded Scala's earlier actor implementation in the standard library. To define an Akka actor, one usually creates a subclass of `Actor`, and implements the `receive` method there:

```
class PingPong extends Actor {
  def receive = {
    case Ping => sender() ! Pong
  }
}
```

The basic thing an actor does is to receive input messages, and process them by doing something to its internal state or sending messages to other actors. However, there's much more features available in Akka. While many of these can be configured via properties or members, there's a range of mixed-in behaviour for special `Actors`:

[25] `http://doc.akka.io/docs/akka/2.3.11/scala.html` (visited on 2015-06-23).

- `Actor with ActorLogging` turns on the default logging mechanism for actor systems, which is useful for debugging.
- `Actor with Stash` extends the default message box behaviour such that messages can be "put aside" currently ("stashed"), and treated again later.
- `Actor with RequiresMessageQueue[BoundedMessageQueueSemantics]` (or with some other semantics) changes the way the message box of the actor is handled; this is a generalization of the behaviour found in `Stash` (which is a subtrait of `RequiresMessageQueue[DequeBasedMessageQueueSemantics]`).
- `Actor with FSM[State, Data]` mixes in another DSL, providing an additional wrapper over `receive` and allowing actors to be defined as Finite State Machines, reacting to input messages and updating their internal state according to the current state.

The last point is an especially interesting one, since this way of providing DSL syntax is very similar to the design used in the practical part of this work: a completely new layer of syntax is mixed in, with new "statements" to be used in the primary constructor, which then internally call the underlying implementation (in the case of actors, `receive`).

From these examples we can see that providing an interface using mixins in this style does not only allow a user to finely choose what behaviour or variation thereof is wanted, but also give them choice over what style of DSL is wanted or needed. A more extreme example of this style is ScalaTest – this library considers itself a "test toolkit", and provides a wide selection of classes and traits to be combined. Its philosophy is to "source out" every aspect of testing into its own module (for example, underlying test platform, kind of testing, specification syntax, documentation creation, and so on), and allow the tester to combine all these freely into what fits them:[26]

```
abstract class UnitSpec extends FlatSpec with Matchers with
  OptionValues with Inside with Inspectors
```

This class does not need a body, all its functionality and style choices are mixed in; it will just be used as a base class for all unit test modules in this project. (A larger example of ScalaTest has already been shown in Listing 2.2.)

[26] Example taken from `http://www.scalatest.org/user_guide/defining_base_classes` (visited on 2015-06-24).

4 Action Systems and Testing: Definition & Background

In the remaining part of this bachelor thesis, the practical work done is described. It consists of a library to program a restricted variant of so-called Action Systems, a formalism similar to state machines or Labelled Transition Systems. The resulting library provides basic functionality to execute such Action Systems using various methods, and, most importantly, a DSL to write them concisely in Scala. As many parts as possible are parametrized and thus left open for more specific use cases or future enhancements. The library has the current working title `actium`, which is the latinized name of the small town of Ἄκτιον in ancient Greek, place of the Battle of Actium – but mostly, a pun on the word "action".

The description of the implementation consists of three parts. First, in this section, an introductory overview of the principles and the usage of Actions Systems and their application in model-based testing is given, to establish a background for the applicability of the library and terms commonly used. Following that, details of the implementation are provided, with a focus on the DSL interface and how its form could be achieved programatically, exploiting the language features of Scala. In the couse of this, also an overview of the functionalities of `actium` is given. Lastly, the finished state of the project is summarized, and the lessons learned are pronounced, which includes notes about the development process, about language features and their usage, and about other directions that could have been followed or were not tried out. This last section also reflects on the state of the library and its limitations, and possible improvements.

Action Systems were introduced to formalize the collaboration of processes in distributed systems [BK83], and proposed as an approach dually to other, more process-focused approaches such as Communicating Sequential Processes (CSP) [Hoa78]. Somewhat informally, an Action System in that original formulation consists of *processes*, each having associated some variables with their initializations, and (named) *actions*, each consisting of a collection of participating processes, a *guard* (predicate), and a statement. An action (or rather, its statement) can be executed if all of its processes are not currently participating in another action, and the guard is satisfied – in that case, the action is said to be *enabled* [BK88]. The order in which the actions are executed, or how they are chosen if there are multiple enabled

```
move(x:MyNat, y:MyNat) if mode == Air && engine == 1 then
{
  pos_x := pos_x + x;
  pos_y := pos_y + y;
};
```

Listing 4.1: Description of an action `move` in a concrete syntax. The action has two parameters, a guard, and a statement consisting of two sub-statements. `mode`, `engine`, `pos_x` and `pos_y` are (global) state variables.

actions, is left unspecified; the system can be executed using different strategies, synchronuously or concurrently.

In the programming work done for this bachelor thesis, a restricted variant of the above formulation is used, which is applied in the context of *model-based mutation testing* [Aic+14]. Mutation testing [Ham77] is a technique to automatically generate a suite of useful test cases for a given system, by systematically injecting small syntactic faults (called *mutants*). A test suite is then generated from the mutants by incrementally adding test cases which can distinguish a mutant from the original implementation. However, to ensure that mutants actually do anything "useful" (and not change the behaviour in a trivial way, or only change dead code), they must first be analyzed in some way, which traditionally involved manual inspection. Mutants with such "useless" behaviour are called *equivalent* (since they are indistinguishable by test cases).

Model-based mutation testing combines this approach with model-based testing: instead of directly operating on the system under test (SUT), a model of it, called a *test model*, is used. This model, commonly described in a symbolic, abstract fashion, is assumed to be correct and usually simpler than the SUT, since only the properties of interest for testing are modelled. It has the advantages that one has more control of the test generation process; for one, it allows to treat the SUT through the model in a purely abstract, black-box manner, which enables non-software systems to be tested as well. Equally important, checking mutants becomes easier, because the mutation process can be resticted to a more controlled set of syntactic mutation operators, and equivalent mutants can be excluded by means of constraint solving. Furthermore, by using such a model, nondeterminism, which can occur both in the SUT or because of the abstractions arising from the modelling process, becomes easier to deal with. A rather complete overview of this methodology is given in [Jöb14].

The point of Action Systems in this environent now is to serve as test models for mutation testing, especially of non-deterministic systems. Their quite simple, formal nature allows to concisely write a model of the SUT, and then transform this system into a symbolic representation. This representation can be relatively easily mutated, and the resulting representations can be passed to a constraint or SMT solver which analyzes the system and the mutants. That such an approach is not only possible, but can also be done efficiently, is shown in [AJT15].

As said above, for this purpose, it is practical to use a more narrowed down form of Action Systems than those defined initially. It turns out that it is possible to

write a system in such a way that it looks similar to a state machine, but keeps the meaning of it as an Action System: we define a set of state variables with their initial values, and a set of labelled actions. Each of the actions can, in addition to the state variables, have parameters. It can also have a guard, involving its parameters and state variables, and a statement, assigning new values to state variables depending on their old values and the parameters. How such an action definition can look like is shown in Listing 4.1, using a syntax very similar to the one defined in [Tap15, p. 16].

In this formulation, the definition of processes is left implicitly; such systems are equivalent to Action Systems in which only one process exists, which also contains all the state variables. The named similarity to a state machine is not an unuseful one, but should be taken with care – the labels of actions should *not* be mistaken for states. Rather, the similarity is of a different nature: the system in such a form is more like a FSM with data path, but only one state. In such a setting, the actions are just labelled transitions from the one state to itself; they only differ in their conditions and update operations. The relevant information for execution of the system is then just the sequence of transition names, not of states. The data-path state is encoded in the state variables; such a system could in principle be transformed into a real state machine, but would soon suffer from state-space explosion (or turns out to be impossible, since the states covered by an Action System are not even necessarily finite).

A more accurate, though also more formal description of Action Systems (in the full original variant as well as the simplified one used here) can be given via Labelled Transition Systems [Tap15, p. 11; Jöb14, p. 21]; this formalism is also used to express formally the conformance relations between different models (test models, or mutated models of one original model), and is used practically as an intermediate representation in various conformance checking algorithms.

The Action Systems which can be described by `actium` are exactly of this nature. The basic style of the syntax is inspired by the one mentioned above; however, some keywords are taken from the Gherking specification language[27], and some things had of course to be changed due to Scala's native syntax. (The idea of using Gherkin-like syntax for a Scala DSL is not new: it exists already at least in the `FeatureSpec` test style of ScalaTest.[28])

[27] `https://github.com/cucumber/cucumber/wiki/Gherkin` (visited on 2015-06-06)
[28] `http://doc.scalatest.org/2.2.4/#org.scalatest.FeatureSpec` (visited on 2015-06-24)

5 `Actium` – a DSL for Action Systems

This section consists of four parts, which try to introduce `actium` to the reader from the lowest to the highest level. The first subsection describes the non-DSL interface and the basic pattern of how an Action System is defined as a conventional class. The second one explains the inner workings of the library, and how systems are executed internally. Finally, the remaining two sections show how the language features of Scala were used to built on top of that the desired domain-specific interface, and describe some interesting implementation details, as well as some additional functionalities.

5.1 Building a System

To explain the features of the implementation on an actual example, a very simple Action System for an imaginary rocket will be used. The definitions and possibilities of `actium` are shown step-be-step using this system. In Figure 5.1, a state diagram of

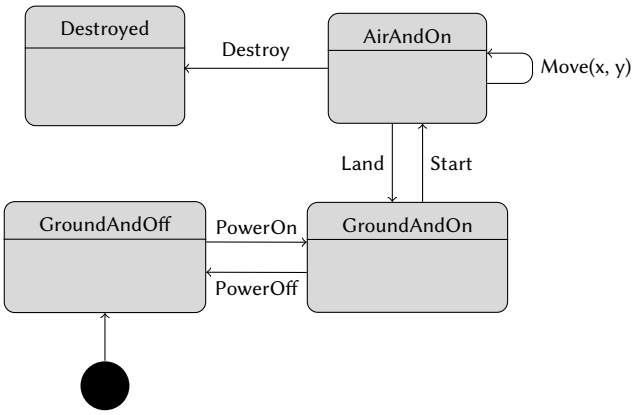


Figure 5.1: State diagram of the `SimpleRocket` Action System. What happens internally, is not specified, and no data path operations are mentioned; the important information is that only sequences of actions (transitions) from this graph are allowed. When the `Destroyed` state is reached, no more actions are possible and the system must stop.

it can be seen. Basically, the rocket can be turned on, start, fly, land, and be turned off; when flying, it can also be destroyed. Note that implementing this as an Action System does not necessarily have to follow the shown partition into states; how internal state is handled is completely up to what form of modelling and refinement is used, and state variables can be chosen as needed. The important information of that diagram is to show the allowed sequences of Actions by the transitions of the graph.

On the most basic level of defining an Action System, the `ActionSystem` trait and Scala's standard syntax are already enough to achive a certain level of expressiveness superior to many other languages. A concrete implementation will typically be an `object`, if all aspects of the behaviour are already fixed, or an abstract class leaving some parts, so that they can later be mixed in on a per-instance basis (especially "choice traits", as `RandomChoice` in the example below – these will be explained in detail in the next subsection). Basically, an `ActionSystem` instance can be created with e.g.

```
val r1 = new SimpleRocket with RandomChoice
```

On this `r1`, we could now, in principle, call the methods `addAction` and `initialize` to add all necessary behaviour; however, the idea is to specify everything possible in the primary constructor of the class or object, since in most cases, the behaviour of an instance is known beforehand (still, the possibility to construct an `ActionSystem` instance programatically, e.g., from an external specification, is left open). Scala makes this construction in the constructor especially easy by treating everything inside a class body as part of the primary constructor; so, if our example class looks like

```
abstract class SimpleRocket extends ActionSystem with GherkinSyntax {
  ⋮
  initialize(something)
  addAction(Action('foo, True, stmts))
}
```

at instance creation the initial environment will be set to `something`, and one action `'foo` be added – there is no need to declare an extra constructor method. In fact, in many cases, an implementation will do without any method declarations whatsoever, by just putting everything in the primary constructor (that is, the class body).

Now, lets consider an actually useful instance for the above example. The following statements are meant to be put directly into the above `SimpleRocket` class. First, we should define how we are going to represent the state of the system; in this case, we encode everything as integers:

```
type State = Int
```

Then, in order to actually write something useful, it is convenient to first give some initializations to the system, declaring shorthands for states and symbolic variables:

```
val F = 0
val T = 1
val Air = 0
val Ground = 1
val Destroyed = 2

val engine = Variable('engine)
val mode = Variable('mode)
val pos_x = Variable('pos_x)
val pos_y = Variable('pos_y)
```

The `Variable` constructor is provided by the library. Some of these definitions are not a strictly necessary part of the system declaration, but will improve readability later. To get a runnable system, we also need to initialize it:

```
initialize(
  Assignment(engine, Constant(F)),
  Assignment(mode, Constant(Ground)),
  Assignment(pos_x, Constant(0)),
  Assignment(pos_y, Constant(0))
)
```

The only task of `initialize` is adding a list of `Assignments` to the environment in order to provide the state variables and their initial values. Again, `Assignment` and `Constant` are part of the symbolic representation provided by the library. Here, only constants are assigned, but in principle, any expression could stand on the right hand side of an assignment. The assignments are evaluated individually, one after the other, at the time of adding (that is, at class construction), and each evaluation is done using the current environment – this allows assigning state variables based on previous values, like

```
Assignment(pos_x, Constant(0)),
Assignment(pos_y, pos_x)
```

where at the definition of `pos_y`, `pos_x` is already existing in the environment.

If all helpers are defined, and everything is initialized, the only thing remaining to do is adding the actual actions. For this purpose, there exists the method `addAction(action: Action): Unit`, which updates the internal table of actions. The exact definition of `Action` will be shown below (Page 57) and contains the name, the condition, the statement, and optionally the parameters of an action. To actually construct an action, it is necessary to put in the ASTs of all these parameters:

```
addAction(Action('PowerOn,
  And(Predicate2("==", engine, Constant(F)),
      Predicate2("!=", mode, Constant(Destroyed))),
  Seq(Assignment(engine, Constant(T))))
)
```

which corresponds to

```
powerOn() if engine == 0 && !(mode == Destroyed) then
{
  engine := 1;
};
```

in the external syntax [Tap15]. For parametrized actions like the one shown in Listing 4.1, we can fill in `Action`'s optional last parameter (which otherwise defaults to `Seq()`):

```
addAction(Action('Move,
  And(Predicate2("==", engine, Constant(T)),
      Predicate2("==", mode, Constant(Air))),
  Seq(Assignment(pos_x, Application("+", pos_x, Variable('dx))),
      Assignment(pos_y, Application("+", pos_y, Variable('dy)))),
  Seq(Variable('dx), Variable('dy)))
)
```

The whole system, defined on this level of the syntax, can be found in Appendix A.2 (for comparison, Appendix A.1 contains the definition in the original external syntax).

There is not much more to be said now about constructing the system; in principle, the shown methods are all which is needed. Also, the term DSLs are not much more complicated than in the examples above: value expressions are constants, variables, or function applications; conditions are predicates on expressions or propositional terms thereof, using `And`, `Or`, and `Not`; and statements are mostly `Assignments` of a variable to an expression.

5.2 Underlying Architecture and Funtionality

The library is built up in as modular a way as possible – what can be parametrized, will be parametrized, and what can be factored out in a trait, will be in a trait. On the other hand, as Scala makes it very easy to introduce small wrappers and ADTs in the form of case classes, these are also frequently used, even if a plain type maybe would have been enough. A rough graphical overview of the most important classes and traits is given in Figure 5.2, which is not really a class diagram, but rather the relations that could be found in an example case for some `ConcreteActionSystem`. Note that the members listed there are not complete, and their types in some cases are simplified; the diagram only tries to capture the essence of the typical hierarchy.

The main part of ACTIUM's functionality is implemented in the trait `ActionSystem`. As indicated by the name, this trait represents the functionality of an Action System, which consists of adding actions, initializing and resetting the state, and executing the system. The respective methods for these abilities are provided by the trait, together with fields containing the necessary state. There are two kinds of things left abstract by `ActionSystem`: the `State` type, and the way actions and parameters are practically chosen at execution.

While most methods of `ActionSystem` are actually just more complex getters or setters, the main logic of execution lies in `run`. This is technically also not really a

method (although it compiles to one); rather, the result of `run` is a recursively defined `Stream[Choice]`. Streams are Scala's standard implementation of lazy collections – they can be potentially infinitely long, since next elements are calculated only on demand. `Choice` is not more than a case class containing the label of an action, and a map of the chosen parameters:

```
final case class Choice[State](label: Label,
  params: Map[Variable[State], State] = Map[Variable[State], State]())
```

In summary, when calling `run`, one has at hand a lazy `Stream` of the stepwise results of the execution of the system (as it is done according to the concrete choice implementation). From that stream, one can "take off" arbitrarily many steps by calling the stream's `force` method (usually the number of taken elements will be restricted in some form, e.g. using `take`):

```
scala> val r = new SimpleRocket with RandomChoice
r: actium.examples.simpleRocket.SimpleRocket with actium.RandomChoice =
  ↳ $anon$1@69005b59

scala> r.run.take(5).force
res0: scala.collection.immutable.Stream[r.Choice] =
  ↳ Stream(Choice('PowerOn), Choice('Start), Choice('Destroy))
```

The actual updates to the state only happen when the elements of the stream are evaluated, and persist afterwards – so, before calling `run` again from the initial state, the system has to be `reset`. With functions like `take` it is possible to only run as many actions as desired, and then continue later with the execution. In the example, there were not even run as many actions as possible, since the execution ended in the final state `'Destroyed` after only three steps.

To define a concrete instance, the abstract members must be provided by the concrete object or class for the system in question. Concrete methods for choosing parameters and actions are in the current implementation usually factored out into *choice traits*, which can be used to mix in various behaviours of choosing; at the moment, there are three useful variants of them: `RandomChoice` uses a uniform random distribution to choose actions and parameters, `IOChoice` provides a console interface for entering all information, and `StaticChoice` is thought mainly for testing, as it allows to fix a sequence of actions and parameters beforehand. Of course, these traits could also be left out totally, and a new, custom way of choice be implemented.

The `State` type is typically defined inside the body of the system object, where the main behaviour is defined using the necessary calls of `initialize` and `addAction`. It is also possible to just define `State` and the actions, and leave the choice abstract, allowing it to be mixed in at a later time. This can be used to treat a system differently, e.g., for testing its correct behaviour and actually using it for running a simulation, or for simply exploring it in the console.

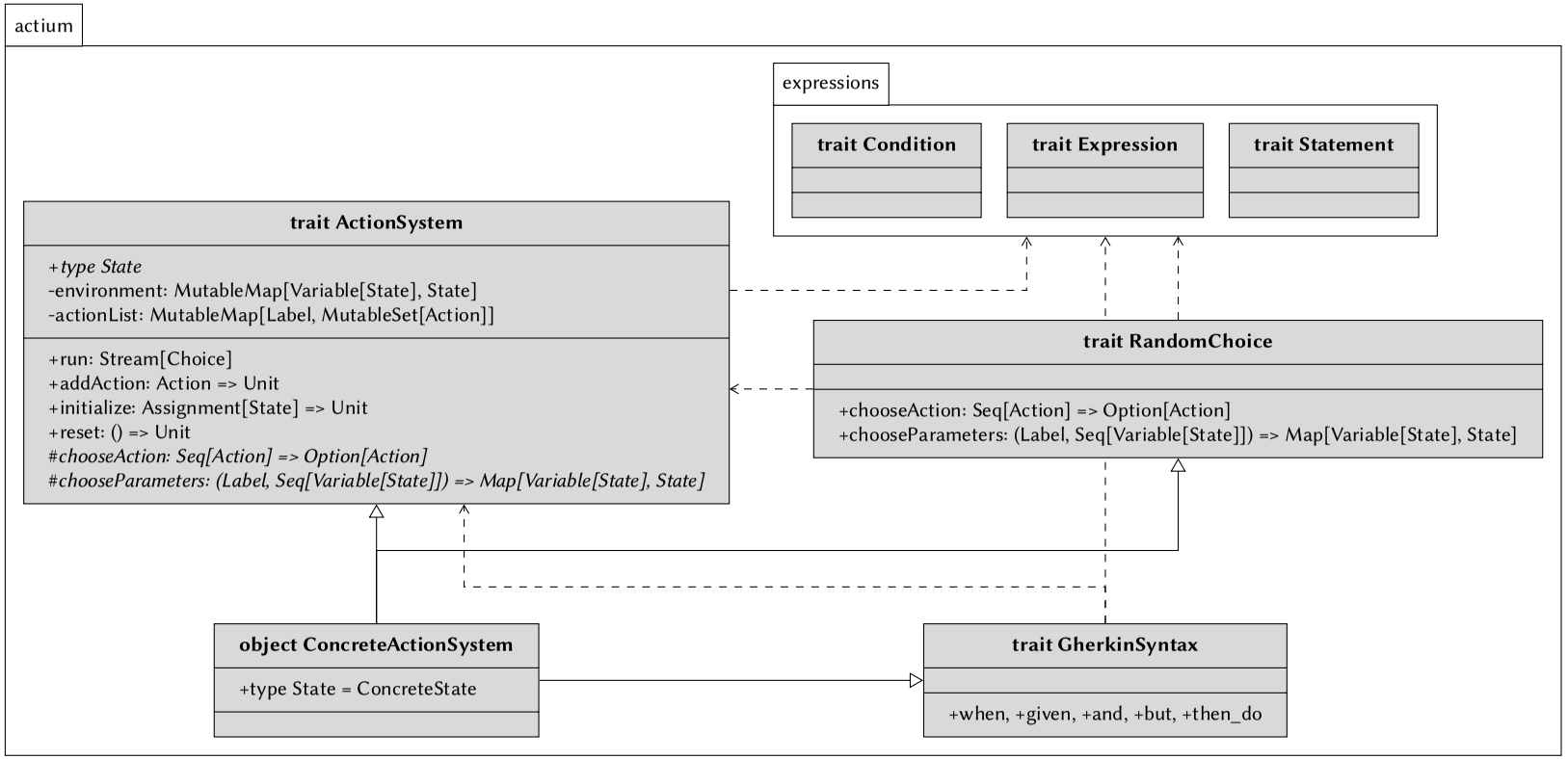


Figure 5.2: Overview of the most important parts of the architecture of `actium`. This is an example configuration for an imaginary concrete Action System. The trait members are not all shown in their precise form; this is just to provide an overview from the implementor's point of view. Any concrete Action System needs to inherit from `ActionSystem` and some implementation of choice, which provides `chooseAction` and `chooseParameters`. Furthermore, the implementation will mix in the DSL syntax from `GherkinSyntax`. All these traits depend on `ActionSystem` by restricting their self types.

The subpackage `actium.expressions` contains the classes for the ADT representation of expressions, statements and conditions (predicates on expressions). The three of these are abstract sealed traits, implemented by case classes for the possible values of each (such as `And` for `Condition`, or `Assignment` for `Statement`). The classes are also all tagged with a `State` type, and the traits contain evaluation of the representations as functions taking an environment and returning a `State` (or a `Boolean`, in the case of `Condition`).

Expressions are the most important building blocks of the behaviour of an `ActionSystem`, since they determine when and how the state changes. The whole representation of an action is in fact defined just like this:

```
final case class Action[State](label: Label,
  condition: Condition[State],
  statements: Seq[Statement[State]],
  params: Seq[Variable[State]] = Seq[Variable[State]]())
```

That describes an action by its label, ADTs for the conditions upon which it gets enabled (these can depend on the state as well as the parameters of the action) and the statements which will be executed in the action, and optionally a symbolic representation of the parameters the action takes. The main part of the definition of an Action System consists of constructing such `Actions` out of small ASTs for statements and conditions, and registering them using `addAction`.

There is not really much DSL-specific to say at this point but the following: since the definition of `Action` shows that an Action System's behaviour is represented symbolically by an AST, `actium` can be called a deep embedding (cf. Page 7). This form of representation was chosen because it does not only allow execution in different ways, but also can be transformed into other formats and used for different purposes than just running a system. For example, there exists a translation function to the AST of the `as2bmc` library [Mad16], which in turn is able to transform Action Systems into logical expressions for the `Z3` SMT solver (see Subsection 5.4). These can be used to perform mutation-testing by constraint solving, as mentioned in the introduction of this part (see Page 48).

5.3 Improving the Syntax

This subsection is concerned mainly with what is happening in `GherkinSyntax`. This trait is an optional mixin for `ActionSystem`, but forms the main point of exploration relevant to this work. For implementing an `ActionSystem` in the DSL style, one can use language features and `GherkinSyntax`' helpers on three levels: basic definitions, action specifications, and expressions.

The main DSL part comes in at taking the methods shown above, and improving the way they are written. Until now, all methods shown are already implemented in `ActionSystem`; now the real DSL part, as implemented in the mixin `GherkinSyntax`, is explained. Its purpose consists mostly of providing methods to replace the above

```scala
sealed trait Expression[+A]
case class Application[+A](op: String, args: Expression[A]*)
  extends Expression[A]
sealed trait Value[+A] extends Expression[A]
case class Constant[+A](value: A) extends Value[A]
case class Variable[+A](name: Symbol) extends Value[A]

sealed trait Statement[+A]
case class Assignment[+A](variable: Variable[A],
  assignment: Expression[A]) extends Statement[A]
case class ExternalAction[A](run: () => ExternalResult)
  extends Statement[A]

sealed trait Condition[+A]
case class And[+A](a: Condition[A], b: Condition[A]) extends Condition[A]
case class Or[+A](a: Condition[A], b: Condition[A]) extends Condition[A]
case class Not[+A](a: Condition[A]) extends Condition[A]
case class Predicate1[A](p: String, a: Expression[A])
  extends Condition[A]
case class Predicate2[A](p: String, a: Expression[A], b: Expression[A])
  extends Condition[A]
case object True extends Condition[Nothing]
case object False extends Condition[Nothing]
```

Listing 5.1: Simplified definitions of all AST classes in the `expressions` sub-package.

literal form of initialization and defining actions by something more natural, ideally in a style very similar to the shown external DSL. Additionally, there are a variety of helpers provided to construct expressions and conditions in a more natural way, looking like native expressions, relieving the need of constructing their ASTs by hand. Both kinds of helpers make heavy use of implicits and extension wrappers.

To understand how the syntax for `actium` was chosen, consider again the following action definition from the external sytax:

```
powerOn() if engine == 0 && !(mode == Destroyed) then {
  engine := 1;
};
```

To translate this into a Scala expression, we can inspect its parts and proper ways of representing them. First, the guard of the action,

```
engine == 0 && !(mode == Destroyed)
```

is just a boolean expression which should, when translated, end up in the following AST representation, as we have seen above:

```
And(Predicate2("==", engine, Constant(F)),
    Predicate2("!=", mode, Constant(Destroyed)))
```

How such expression syntaxes can easily be embedded into Scala by the help of operators has been shown in Subsection 2.1. Because we are dealing essentially with predicate logic, the definition of these operators spreads over two places: `Expression` and `Condition`. In `Condition`, only the boolean combinators such as `&&` or `||` are defined, whereas predicates live in `Expression` (since they operate on values and just

return booleans). It should be noted that, because of issues concerning covariance of the `Expression` trait when constructing a `Predicate2` with the same type parameter, the respective operators are defined not directly in the trait, though, but in an implicit wrapper; there, for example, we have the following definitions:

```
def ===(other: Expression[A]): Condition[A] =
  Predicate2("==", expr, other)
def =!=(other: Expression[A]): Condition[A] = !(expr === other)
```

The use of `===` instead of the more natural `==` is required, because the latter is already defined in `Any` and deepley nested into the language, and thus is not overrideable in a practical way. The choice for `=!=` instead of `!==` was of a different kind: here, one of the exceptions of the precedence rules for operators (Page 11) comes into play, namely, that an operator ending in `=` is treated as an assignment operator – *unless* its name also starts with `=`, in which case it is treated like a comparison operator.

With the help of these operator methods, we are already able to write things like

```
Variable('engine) === Constant(0)
```

What remains is to also get rid of the explicit constructors for constant literals. We might try to achieve this by providing the according implicit conversions:

```
implicit def literalToConstant(s: State): Constant[State] = Constant(s)
implicit def symbolToVariable[A](name: Symbol): Variable[A] =
  Variable(name)
```

Unfortunately, this does not really work out using the current setting. The reason is that to be able to write

```
'engine === 0
```

two implicit conversion would have to be called: one from `Symbol` to `Variable`, and one for the wrapper for `Expression` containing `===`.

This problem can be resolved in three ways: on the one hand, one could change the place in which `===` was defined to avoid the double wrapping – this can be done by implementing `===` directly in `Expression`, or by additionally providing a wrapper for `Symbol` which contains `===` and performs the wrapping in `Variable` automatically. Both of these have disadvantages, though: making `===` part of `Expression` enforces defining the trait's type parameter for `State` to be invariant, which might lead to future complications when introducing better support for more complex `State` types; and implementing a double wrapper for `Symbol` will lead to more complex definitions, while it does not even solve the problem fully, as using too many implicit conversions on interferring levels leads to unexpected complications of them.

On the other hand, there is a very simple, although not that succint solution, which has been chosen in the current implementation and the examples: we just do not use an implicit conversion for the left hand side of a call to `===`. Instead, we can define the state variable `'engine` as a `val` in the scope of the class; this has already been shown in the previous subsection:

```
val engine = Variable('engine)
```

In order to make this definition a bit more linguistically applealing, there is provided a simple method

```
def statevar(name: Symbol) = Variable[State](name)
```

allowing to write instead

```
val engine = statevar('engine)
```

(Although this is arguably not too big an improvement; we might instead wish to get rid of the redundancy of mentioning the name of the state variable twice, but this is currently only possible by using experimental, non-standard macro facilities such as macro annotations.)

Now that these operators and conversions are in place, we finally are able to write

```
engine === F && mode =!= Destroyed
```

whereby `engine` and `mode` are `Variables` defined as `vals`, and `F` and `Destroyed` are just names for the ints `0` and `2`, in this case.

Assignment statements and value expressions work in a similar way to guards/`Conditions`. The operator `:=` is just a method of `Variable`, taking an `Expression` and returning an `Assignment`. In this case, the operator has an intuitive name, and its precedence is as expected. The right hand side of this operator can be an arbitrary expression. Expressions themselves are either `Variables` (containing a symbol for the name), `Constant` (containing a literal value of type `State`), or `Applications` of one operator (represented by its name) to a list of other expressions.

Given these basic building blocks (expressions, conditions, and statements) and their symbolic (DSL) representations, we are finally ready to look into the construction of action specifications such as this:

```
when('Destroy) given mode === Air then_do (
  mode := Destroyed,
  engine := F
)
```

Converted to the underlying AST translation, this should look like the following code:

```
addAction(Action('Destroy,
  Predicate2("==", mode, Constant(Air)),
  Seq(Assignment(engine, Constant(F)),
      Assignment(mode, Constant(Destroyed))))
)
```

To be able to achieve this in Scala, the trait `GherkinSyntax` introduces the "keywords" `when`, `given`, and `then_do` (as mentioned above, the first two names were taken from Gherkin). These keywords are all methods called in dotless style, and work by repeatedly updating information in immutable builder objects. These are keeping track of the data provided, which finally gets used in `then_do` to call `addAction` of

the `ActionSystem`. For example, the above definition (`when('Destroy) ...`) can be reduced in one step to the following:

```
new WithLabel {
  def label = 'Destroy
  def params = Seq()
}.given(Predicate2("==", mode, Constant(Air)))
 .then_do(Seq(Assignment(engine, Constant(F)),
              Assignment(mode, Constant(Destroyed))))
```

The `WithLabel` builder is the actual result of the `when` method, and contains the name and optionally the parameters of the action. `given` is a method of `WithLabel`, which in its body constructs a new builder with more information, namely, the condition, and notes this information in the result type; therefore, the above reduces further to this:

```
new WithLabel with WithCondition {
    def label = 'Destroy
    def params = Seq()
    def condition = Predicate2("==", mode, Constant(Air))
}.then_do(Seq(Assignment(engine, Constant(F)),
              Assignment(mode, Constant(Destroyed))))
```

We now have an object of type `WithLabel with WithCondition`. For this type, an implicit wrapper is defined, containing the final `then_do` method, taking the list of statements as variadic parameter list. It uses the accumulated information, puts it into an `Action` object, and adds it to the system; inlining the accumulated data into `then_do` leads to this final result:

```
 { val a = Action('Destroy, Predicate2("==", mode, Constant(Air)),
               Seq(Assignment(engine, Constant(F)),
                   Assignment(mode, Constant(Destroyed))),
               Seq())
   addAction(a) }
```

which is exactly equivalent to the translation of the example we began with.

This describes in the most basic form the idea behind the definition statements. Actually, the situation is more complicated; this is why there are different types of builders and extensions involved. The reason for the complication is to allow the user a certain freedom in the order and choice of the keywords. For example, the DSL allows also the following to be written:

```
given (mode === Air) when 'Land then_do (
  mode := Ground
)
```

In this case, the order of `given` and `when` is turned around. Or, we could also leave out the condition, if the guard is trivial, like in this hypothetical action:

```
when('OpenWindow) then_do (
  window_state := Open
)
```

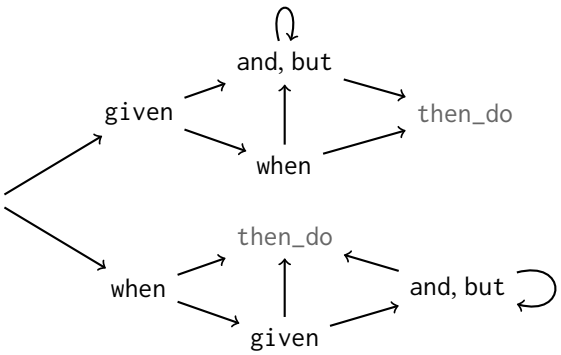


Figure 5.3: Hierarchy of valid command sequences. The highlighted `then_do` always concludes a sequence; by applying it to a list of statements, the actions gets defined and added to the system.

The problem is that in a naive implementation of the keyword methods, `then_do` could be called on a builder object in an invalid state; for example, if it were defined on the object returned by `given`, we could define the following action:

```
given(window_state === Open) then_do (
  temperature := Cold
)
```

which is not a proper action definition, since it lacks the label.

To ensure that only valid sequences of keywords can be used to build up an action, we first fix the allowed keywords and their order. This is done in Figure 5.3. Given that hierarchy, and the knowledge that there are two relevant parts of information expressed by `given` and `when` (the label and the guard of an action), a type-safe chain of calls can be formed. The necessary setup to enforce this is listed in Listing 5.2 (in a truncated way, since only the interface is relevant here).

To achieve the restriction of call sequences, two methods `given` and `when` are defined at top level, each returning the respective builder object, extended with the possibility to call the "opposite" method via the traits `Given` and `When`. By having the methods `given` and `when` twice, and in separate place, it is ensured that each of them can only be called once, but in any order. Both methods store the particular information collected at the point of their calling in the partial builder traits `WithLabel` and `WithCondition`. At last, for `WithLabel` and `WithLabel with WithCondition`, implicit wrappers are provided implementing the final calls to `then_do`.

In addition to the methods shown in the listing, the wrappers for `WithCondition` and `WithLabel with WithCondition` also contain the methods `and`, `or`, and `but`, which can be used as synonyms for `given` to add further conditions in a natural-language fashion – their only functionality is to produce a conjunction or disjunction of the `Condition` on the left and on the right; for example, the definition

```
when('PowerOn) given engine === F but mode =!= Destroyed then_do (
  engine := T
)
```

```
def when(lbl: LabelWithParams): WithLabel with Given = ???
def given(cond: Condition[State]): WithCondition with When = ???

trait WithLabel {
  def label: Label
  def params: Seq[Variable[State]]
}

trait WithCondition {
  def condition: Condition[State]
}

trait When { self: WithCondition =>
  def when(lbl: LabelWithParams): WithLabel with WithCondition = ???
}

trait Given { self: WithLabel =>
  def given(cond: Condition[State]): WithLabel with WithCondition = ???
}

implicit class WithLabelAndConditionWrapper(
    builder: WithLabel with WithCondition) {
  def then_do(stmts: Statement[State]*): Unit = ???
}

implicit class WithLabelWrapper(builder: WithLabel) {
  def then_do(stmts: Statement[State]*): Unit = ???
}
```

Listing 5.2: The trait setup to ensure that only valid sequences of statements can be used in an action definition. These methods and traits are all defined in `GherkinSyntax` and mixed in with it. The implementations are shown as unimplemented here (???).

would be equivalent to simply

```
when('PowerOn) given engine === F && mode =!= Destroyed then_do (
  engine := T
)
```

The only thing that remains to be explained now is how action parameters are allowed to be specified in the convenient call-like fashsion looking like

```
when('Move('dx, 'dy))
```

The solution is relatively simple: the argument type of `when` is not `Symbol`, but actually `LabelWithParams`, a small wrapper containing the label and optionally the parameters:

```
case class LabelWithParams(label: Label, params: Seq[Variable[State]])
```

To be able to just use symbols and normal application, the following implicit conversions are in scope:

```
implicit def symbolToLabel(s: Symbol): LabelWithParams =
  LabelWithParams(s, Seq())
```

```
implicit class LabelWithParamsBuilder(s: Symbol) {
  def apply(params: Variable[State]*): LabelWithParams =
    LabelWithParams(s, params)
}
```

In principle, we are now able to fully emulate the original syntax for Action Systems. Some more intricate details of implicit conversion and syntactic tricks have been left out, but the basic principles should have been made clear. The full `SimpleRocket` example at this point is listed in Appendix A.3.

5.4 Extensions to the Basic Functionality

Finally, besides the basic DSL framework described in the last section, there have been implemented some small extensions. Two of these concern the semantics, as compared to the existing implementation, while the third one deals with enabling integration of this DSL into another existing framework, which forms the basis for a mutation testing pipeline.

The first one of these is quite simple: `actium` allows to define multiple actions with the same label; that is, using `when('Something)` multiple times will not cause any problems. This is handled internally by simply associating with each label not an `Action`, but a `Set[Action]`, and flattening the sets accordingly at evaluation. The individual actions are, however, always properly distinguished when needed; for example, when running the system with a choice trait, or when converting a system to an alternative symbolic representation.

The second semantic extension is the introduction of an additional statement besides assignment, which can not be found in other Action System formulations: `externally`. This statement is defined in the extra trait `ExternalEffects` (which is a mixin to `GherkinSyntax`), and allows to embed arbitrary Scala code in an action; this code will be executed each time the action is executed. In the simplest form, this can be just a debug message:

```
when('Destroy) given mode === Air then_do (
  mode := Destroyed,
  engine := F,
  externally(println("BOOM!"))
)
```

The content of `externally` is a closure (represented internally as `() => Unit` and passed by-name), and can thus access external state in the form of `var`s defined in the class, or even through library calls. However, what's even more powerful is that there is full access to the systems state variables and the action's parameters at the point of executing the external closure – which is somewhat more complex to realize. Concretely, we can have an action like this:

```
when('Position('x, 'y)) given mode === Air && engine === T then_do (
```

```
implicit class ValueWrapper(s: Symbol)(implicit env: Map[Symbol, Int]) {
  def value = env(s)
}

case class ExternalAction(block: () => Unit)

def externally(block: => Unit) = ExternalAction(() => block)

implicit var environment = Map[Symbol, Int]()
def then_do(as: ExternalAction*) = new {
  def execute(env: Map[Symbol, Int]) = {
    environment = env
    as foreach (_.block())
  }
}
```

Listing 5.3: A simplified example of how the variable capturing in external statements is implemented: by capturing an implicit reference, and setting it later to the then current environment. `externally` and `ValueWrapper` are in reality provided by the mixin `ExternalActions`; `ExternalAction` is defined in the statement ADT.

```
  pos_x := 'x,
  pos_y := 'y,
  externally {
    println(s"x = ${'x.value}, pos_x = ${'pos_x.value}")
  }
)
```

(whereby `pos_x` is part of the state variables if the `SimpleRocket` system).

What is important here is the usage of the form `'x.value` to access variables from the scope of the action. The method `value` is defined as an extension for `Symbol` – and it takes as an implicit parameter an `ImplicitEnv[State]`. This class is just a wrapper about a `Map[State]`. The fact exploited to make this work is that the actual implicit value, that is provided for this parameter, is defined as a `var`. That way, at each evaluation the environment can be filled with the current values. For better understanding, look at the simplified variant of the pattern in Listing 5.3. There the structure of the relevant commands have been "rebuilt", in a way to resemble the original implementation. It is important to note that `execute` already closes over the reference to the empty environment.

If we now use this example framework to define an action, like so:

```
val pseudoAction = then_do(
  externally {
    println('y.value)
  }
)
```

the reference to `environment` also gets passed implicitly to `value`, and is therefore used in `ValueWrapper`'s definition to look up the value of `'y`. However, since all this is passed by-name, the lookup will not happen immediately and use the empty dictionary; only when we execute the pseudo-action, like this:

```
pseudoAction.execute(Map('x -> 1, 'y -> 42))
```

the closure will actually be called – but before calling, the environment reference will be updated according to the parameter of the `execute` method, which in the case of the actual `ActionSystem` resembles the current state and the action parameters. As a result, the value `42` will be printed out, being declared as `'y`'s value only after the external effect has been defined in the code.

Furthermore, in order to make this statements usable for actual testing scenarios, `externally` supports aborting the execution of the system. For this purpose, there exists a member

```
def abort: Nothing = throw AbortExternallyException
```

and `externally` is, in fact, not only a trivial wrapper around a thunk, but catches this exception and "transforms" the result:

```
def externally(block: => Unit): ExternalAction[State] =
  ExternalAction(() => {
    try {
      block
      Succeeded
    } catch {
      case AbortExternallyException => Failed
    }
  })
```

Therefore, also the result of the closure used in `ExternalAction` is not `Unit`, but `ExternalResult`, which can be either `Succeeded` or `Failed`. This result is checked in the `run` method of `ActionSystem`, and, if negative, stops it from producing further values for the `Stream[Choice]`.

This possibility should help when, for example, using external statements to do side-by-side testing of an underlying SUT. One could imagine to use an Action System to simulate expected behaviour, and at every step comparing its modelled output to the real system; if a erroneous difference occurs, the execution can be halted immediately. A modification of the SimpleRocket example, involving examples of external statements, is given in Appendix A.4.

Using exceptions in such a way, namely, to introduce non-local control flow, might be considered almost an abuse by some people – this estimation is, however, largely depending on one's background. For example, Scala, unlike many other languages, does not have a `break` statement for loops. Instead, a very similar construct to the above is used in the standard library to implement `scala.util.control.Breaks` – providing a *library* function implementing `break`[29]. This power of exceptions to implement non-local control flow has been explored in [Lil99], where a variant of them is shown to be equally powerful to the `call/cc` operator of many LISPs, and is also marginally discussed in [TaPL02]. It can be considered very useful for DSLs in situations like this – if used wisely.

[29] `http://www.scala-lang.org/api/current/index.html#scala.util.control.Breaks` (visited on 2015-06-30)

```
trait As2BmcTranslator { self: ActionSystem { type State = Int } =>
  def toAs2Bmc: ast.ActionSystemTy = ???
  private def translateExpression(
    expr: exp.Expression[State]): ast.ExpressionTyped = ???
  private def translateCondition(
    cond: exp.Condition[State]): ast.ExpressionTyped = ???
}
```

Listing 5.4: Overview of the body of `As2BmcTranslator`. The actual implementations are left out.

As a third enhancement to the original functionality, there is provided a mixin which allows to translate the symbolic representation of the action system to another representation, which might be used by an external library (for example, to actually perform mutation testing on the defined system). As previously mentioned, there is one translator to the `as2bmc` library [Mad16]. This functionality is provided in a trait `As2BmcTranslator`, having a self type of `ActionSystem { type State = Int }`, which is a refinement type ensuring that the trait can only be mixed in to `ActionSystem`s with a `State` type of `Int` – the reason for this being that currently, `ActionSystem` does not have a sufficient handling of arbitrary types, and integers are in most cases enough to encode all necessary information of a system.

An overview of the translation mixin's body is listed in Listing 5.4. There, it can be seen that there is only one relevant method, `toAs2Bmc`, returning the AST of an action system in the "foreign" representation. The other two methods are used only internally to translate the respective parts of the expression syntax, as indicated by their names; their implementation is quite mechanical, since both representations are quite similar, and both consist only of a `match` statement. For example, the first three lines of `translateExpression` are

```
case exp.Variable(v) => ast.VariableExpTy(v.name.toString, stateType)
case exp.Constant(c) => ast.NumConstantExpTy(c, stateType)
case exp.Application("+", a, b) =>
  ast.BinOpExpTy(ast.Addition, translateExpression(a),
                 translateExpression(b), stateType)
```

As indicated, the translation of the basic AST is rather trivial, since both representations store the same fashion of Action Systems in a similar structure; the one thing to be improved is the handling of types. They are represented internally via tagging in `as2bmc`, whereas `actium` currently uses a path-dependent type in Scala, and can currently only be practically used for very simple settings (mostly integers).

6 Résumé: What Can Be Learned From This

This section shall reflect on the practical part, with respect to the questions stated in the introduction: how do the described language features help with real programming problems, which of them are most useful, and what is missing? What are advantages and disadvantages of the taken approach? Where could it be improved, and what are possible future directions? How did the development process took course, and what can be learned from the implementation and its design?

In general, it can be said that Scala fulfilled most of the expectations about its syntax and semantics' abilities. Comparing the original syntax:

```
powerOn() if engine == 0 && !(mode == Destroyed) then
{
  engine := 1;
};
```

with the finally achieved one:

```
when('PowerOn) given engine === 0 && mode =!= Destroyed then_do (
  engine := 1
)
```

we see that a well matching analogy could be created, which would probably be understood right away by someone used to the external syntax. That this matching is quite accurate can also be observed by comparing the full original example in Appendix A.1 with its translation in Appendix A.3.

Furthermore, a few useful extra features have been implemented, such as a bit more flexibility in the keyword usage (`given` and `when` can be swapped, `and`, `but` and `or` used for building the guards, and multiple actions given the same label), or the addition of external statements to allow useful mixing between the system behaviour and Scala code. The feasibility of integrating the library with an existing toolchain has not been practically tested, but a translation of the symbolic representation of `actium` Action Sytstems to the format used by `as2bmc` has been provided with little effort. Since a fully symbolic approch is used anyway, other translations should also not really be a problem.

The process of developing the library went through multiple steps. At first, a kind of top-down approach was tried: this consisted of only defining an interface, which should resemble the "look and feel" of the existing syntax, and contained just dummy implementations. Using this syntax, a few difficulties and dead ends could already be ruled out, and most of the concrete keywords and the "style of writing" were fixed (like the definition of all actions within the primary constructor of a subclass of `ActionSystem`, or the idea of using Gherkin's keywords `given` and `when`). Still, some syntactic subtleties of the DSL and some practical needs could not be identified at this stage.

Therefore, when it came to actually implementing the functionality of running Action Systems, the previous code needed to be rewritten completely, leaving behind only its main ideas. The reason for this was that it turned out to be much easier to try out different variants of expressing domain specific constructs when a working system was already given – this is because then the constructs can immediately be tested for fitting the underlying system, and for not interfering with each other later. With that insight in mind, at first, an almost completely working non-DSL implementation of `ActionSystem` was written, which essentially stayed the same until the final version. On top of this underlying implementation, the additional mixin `GherkinSyntax` was provided, building on the few underlying constructs provided by `ActionSystem` in the way described in Subsection 5.3.

This turned-around approach proved much more practical. While for a smaller, more strictly defined target DSL syntax, it might actually be easier to start at the "top" of the library and then implement out dummy interfaces, until an underlying implementation is completed, the more bottom-up way of layering the convenience syntax above a predefined simple, conventional, and working functionality actually turned out to be more fruitful. In retrospection, having such a separation is also more desirable from architectural and maintainance points of view; for it allows to separate concerns much better, leads to less coupling, and probably facilitates testing (although, honestly speaking, since the whole development was of a rather experimental nature, these concerns were not much considered and often neglected in praxis – but this does not relativize named points). The a-posteriori introduction of other, not primarily considered DSLs (for example, an external variant), or stacking multiple ones, is also enabled and simplified in this way.

Concerning the advantages of Scala's syntax, a certain dichotomy can be observed. On the one hand, it almost everything that was wanted to be expressed could be implemented somehow. In this respect, the ease of defining ADTs with custom operators was of much help; but especially implicits turned out to be the rescue to many of the more intricate problems (so, for example, the solution to "lazy capturing" of state variables described in Listing 5.3). Their universal applicability to interfere almost anywhere in an expression, and in a way transform the types of values there, is a unique and highly beneficial concept.

On the other hand, there were some problems that could not be solved (or rather, some desired syntax that could not be expressed) by standard means. The primary

example of this is the need to put the statements of an action in round parentheses, separated by commata, instead of a block and thus looking like a language statement. This is due to the fact that `actium` needs to use a deep embedding (to be able to operate on its underlying representation), which requires passing every "syntactic form" of the DSL as a Scala expression. Now, if a block were used to pass statements into `then_do`, that would always evaluate to only the last expression in its body. By defining `then_do` to take a variadic parameter list, this can be resolved, at the cost of a less intuitive syntax. This problem could in principle be solved by using a macro for `then_do`, which takes the block argument as an expression, splits it into the individual statment expressions, and puts them in a list; however, as mentioned, practical macro implementations are outside the scope of this work.

Another feature of Scala's syntax has a sometimes unintuitive behaviour as well: the dotless calls used for `given` and `when`. Concretely, due to the way the parser is working and how the resulution of dottless calls is defined, they cannot be interrupted by newlines; that is, writing

```
when('PowerOn)
given engine === 0 && mode =!= Destroyed then_do (
  engine := 1
)
```

would not work, since the `given` in the second line is not recognized as a continued invocation of the result of `when`, but as another statement. This is a basic limitation of Scala, and common knowledge, but can be unexpected from the thinking perspective of a DSL user. It is not a new problem when using a deep embedding; for example, the internal DSL variant of `sbt`'s configuration syntax (used for automatizing Scala builts, somewhat like `make`) suffers from the same limitation [SBT15].

In a similar fashion, though not as much striking, is the in some ways complicated definition of operator precedence used by Scala. In many cases, defined operators tend to "just work", because the most common symbols are assigned their "usual" precedences, and sensible exceptions are made for some special cases (such as recognizing `:=` as an assignment operator). However, sometimes, a certain operator name is desired to be used in a certain position, but will not do the right thing there without parentheses, because it associates in wrong ways. This was the case when, at first, the inequality operator for values was defined as `!==`; the redefinion as `=!=` does the right thing due to another exception rule, but noting this, and having to think about it for every operator, can be frustrating for both developer and DSL user. An explicit possibility to manually define precedences, as in some other languages with operators, would be desireable here (like Haskell's `infixl`/`infixr` declarations).

When it comes to the expressiveness of the Action Systems definable by `actium`, there are some limitations compared to the original implementation. This is due to the fact that this work's main concern was to explore the limits of Scala for DSL implementation; some additional features present in other languages based on Action Systems were therefore left out.

For one, while the `ActionSystem` trait is parametrized by the type `State`, this is currently not of too much use, and `State` is in all examples and tests defaulted or constrained to `Int`. While, with some further work, the DSL could be adapted to some custom state type (this requires syntax and semantics for values, expressions, and predicates on it), this quite complicated (if not impossible with standard features) to be done in general, in a satisfying way. The reason for this is, again, that we are working with a deep embedding – which means that type information needs to be present symbolically in the representation at runtime. On the other hand, the implementation is currently layed out to support static type checking in Scala, which is in principle desirable (since it allows to prevent certain errors at compile time).

The difficulty now lies in the fact that to get all desired features, one would need to represent types both symbolically in the AST and statically in Scala. This could theoretically be done by using a more complicated encoding (using type tags, singletons, or some other technique relying on more advanced typing techniques), but it is unclear whether such an encoding could be done without breaking the current syntax, and without having to require the definer of a type to write too much unnecessary boilerplate again. It would maybe be possible to achieve a satisfying result using macros, again. (The automatic derivation of generic type representations in Shapeless[30] looks primising for this purpose.)

Furthermore, there are some syntactic constructs which `actium` currently lacks. For example, many concrete implementations using Action Systems allow to define actions in a nested way (that is, to define additional guards with different statements inside of one guard); adding support for these would mainly consist of adding a syntactic transformation, and adapting the DSL to behave properly when the keywords are used not at top level.

Moreover, there has also been come up with the idea of adding a kind of pattern matching support for action parameters. This would allow to make the writing of actions with similar guards easier and more readable, especially combined with some sort of wildcard pattern. Implementing this would require extending the parameter-passing mechanism a bit, and, most importantly, an algorithm for unification and ordering patterns by specificity. Whether implementing a reasonable variant of the latter can be done easily, without having to resort to, e.g., an external solver, is unknown to the author.

Besides these enhancements of functionality, it would also be worthwhile to examine and enhance the coherence of the implementation with some formal execution semantics. For instance, currently, the order of execution of statements in an action is somewhat undefined (in practice, they are always executed sequentially). But since Action Systems are a means of formalizing concurrent systems, being able to exactly specify or know execution order would be a desirable property. It might also, even in the current variant, sometimes be unclear how, for example, the side effects of two external statements will be sequenced and influence each other.

[30] `https://github.com/milessabin/shapeless/wiki/Feature-overview:-shapeless-2.0.0#generic-representation-of-sealed-families-of-case-classes` (visited on 2015-07-03)

There is also an issue arising from the frequent use of traits and mixins, which in principle turned out to be a very practical pattern to separate concerns. While traits mostly allow to specify several degrees of coupling very exactly, they are sometimes still too weak to describe what is wanted to be achieved. This limitation was most obvious for the choice mixins, which define how parameters and actions are chosen during execution of an Action System. There, it soon became visible that while mixins are very practical for adding orthogonal static implementations, they are not so easy to use for configuring behaviour. Concretely, at the moment, one can only define an `ActionSystem` with certain choice behaviour by using the `with` syntax; but it often would be nice to allow finer configuration, such as setting specific weights for actions in the `RandomChoice` trait. Here, one could either investigate ways of elegantly allowing parametrization of mixin behaviour, or think about another alternative not using mixins in this way at all (such as a conventional design pattern). Reflecting this, we can conclude that traits are extremely useful for orthogonal static, but less so for runtime parametrization.

Finally, the current implementation is a bit unsatisfactory from a "purist" functional programming point of view. This is because all settings, definitions, but also the execution of systems themselves are based on mutable updates. While this is not a major flaw, it would be desirable to rewrite as much as possible of the execution to using immutable updates – this would enhance some minor points, such as allowing easier pesistance of immediate states, or removing the necessity of resetting systems between test runs. More immutability would also perceived as more elegant by the author, and could lead to more correct code, as errors in mutable updates tend to be harder to find. The best way to introduce immutability would probably be to run an `ActionSystem` like a Mealy machine, or similar to a state monad (copying new state, instead of updating). On the other hand, there is little that can be done to remove mutable updates from the action definition syntax (the `given`/`when` statements in the primary constructor); but this seems more acceptable, as it is done only once at object initalization (at least one could revisit the visibity of `addAction`, and make it as private as possible).

Bibliography

Note: features to be included in Scala are first proposed in Scala Improvement Proposals (SIPs), collected at `http://docs.scala-lang.org/sips/index.html` while they are reviewed or "pending". The SIPs cited here are always noted as "pending" or "accepted" as of the state at the time of writing this (May/June 2015).

[SICP96] Abelson, H.; Sussman, G. J. *Structure and interpretation of computer programs*. 2nd ed. Cambridge: MIT Press, 1996.

[AJ09] Aichernig, B. K.; Jifeng, H. "Mutation testing in UTP". In: *Formal Aspects of Computing* 21.1–2 (2009), pp. 33–64. DOI: 10.1007/s00165-008-0083-6.

[AJT15] Aichernig, B. K.; Jöbstl, E.; Tiran, S. "Model-based mutation testing via symbolic refinement checking". In: *Science of Computer Programming* 97 (2015), pp. 383–404. DOI: 10.1016/j.scico.2014.05.004.

[Aic+14] Aichernig, B. K. et al. "Killing strategies for model-based mutation testing". In: *Software Testing, Verification and Reliability* (2014). DOI: 10.1002/stvr.1522.

[BK83] Back, R. J. R.; Kurki-Suonio, R. "Decentralization of process nets with centralized control". In: *Proceedings of the second annual ACM Symposium on Principles of Distributed Computing (PODC)*. ACM, 1983, pp. 131–142.

[BK88] Back, R. J. R.; Kurki-Suonio, R. "Distributed cooperation with action systems". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 10.4 (1988), pp. 513–554. DOI: 10.1145/48022.48023.

[BO13] Burmako, E.; Odersky, M. *Scala macros*. Tech. rep. École Polytechnique Fédérale de Lausanne, 2013. URL: http://infoscience.epfl.ch/record/183862/files/2012-07-09-MetaPaper.pdf (visited on 2015-06-09).

[Cop95] Coplien, J. "Curiously recurring template patterns". In: *C++ Report* 7 (2 1995), pp. 24–27.

[Dmi04] Dmitriev, S. "Language oriented programming: the next programming paradigm". In: *JetBrains onBoard* 1.2 (2004), pp. 1–13. URL: http://www.onboard.jetbrains.com/articles/04/10/lop (visited on 2015-05-25).

[GW14] Gibbons, J.; Wu, N. “Folding domain-specific languages: deep and shallow embeddings (functional pearl)”. In: *SIGPLAN Not.* 49.9 (2014), pp. 339–347. DOI: 10.1145/2692915.2628138.

[Gos+13] Gosling, J. et al. *The Java® language specification, Java SE 7 edition.* Oracle, 2013. URL: http://docs.oracle.com/javase/specs/jls/se7/html/index.html (visited on 2015-04-23).

[HZ13] Haller, P.; Zaugg, J. *SIP-22 – async.* (Pending). 2013. URL: http://docs.scala-lang.org/sips/pending/async.html (visited on 2015-05-17).

[Ham77] Hamlet, R. “Testing programs with the aid of a compiler”. In: *IEEE Transactions on Software Engineering* SE-3.4 (1977), pp. 279–290. DOI: 10.1109/TSE.1977.231145.

[HB90] Hammond, K.; Blott, S. “Implementing Haskell type classes”. In: *Functional Programming.* Workshops in Computing. Springer, 1990, pp. 265–286. DOI: 10.1007/978-1-4471-3166-3_18.

[HIW10] Havelund, K.; Ingham, M.; Wagner, D. *A case study in DSL development.* Scala Days 2010, 2010. URL: http://www.havelund.com/Publications/scala-days-2010-dsl.pdf (visited on 2015-03-26).

[Hoa78] Hoare, C. A. R. “Communicating sequential processes”. In: *Communications of the ACM* 21.8 (1978). DOI: 10.1145/359576.359585.

[JavaAPI] *Java™ Platform, Standard Edition 7: API specification.* Oracle. URL: http://docs.oracle.com/javase/7/docs/api/index.html (visited on 2015-04-23).

[Jöb14] Jöbstl, E. “Model-based mutation testing with constraint and SMT solvers”. PhD thesis. Graz University of Technology, 2014.

[LM01] Leijen, D.; Meijer, E. *Parsec: direct style monadic parser combinators for the real world.* Tech. rep. Departement of Computer Science, Universiteit Utrecht, 2001. URL: http://dspace.library.uu.nl/handle/1874/2535 (visited on 2015-06-09).

[Lil99] Lillibridge, M. “Unchecked exceptions can be strictly more powerful than call/cc”. In: *Higher-Order and Symbolic Computation* 12.1 (1999), pp. 75–104. DOI: 10.1023/A:1010020917337.

[Lin+13] Lindholm, T. et al. *The Java® Virtual Machine specification, Java SE 7 edition.* Oracle, 2013. URL: http://docs.oracle.com/javase/specs/jvms/se7/html/index.html (visited on 2015-04-23).

[Mad16] Maderbacher, B. “Weak mutation analysis of action system models using bounded model checking via SMT solving”. Bachelor’s thesis. Graz University of Technology, 2016. Unpublished.

[McC60] McCarthy, J. “Recursive functions of symbolic expressions and their computation by machine, part I”. In: *Communications of the ACM* 3.4 (1960), pp. 184–195. DOI: 10.1145/367177.367199.

[MHS05] Mernik, M.; Heering, J.; Sloane, A. M. "When and how to develop domain-specific languages". In: *ACM Computing Surveys (CSUR)* 37.4 (2005), pp. 316–344. DOI: 10.1145/1118890.1118892.

[MOH13] Miller, H.; Odersky, M.; Haller, P. *SIP-21 – spores*. (Pending). 2013. URL: http://docs.scala-lang.org/sips/pending/spores.html (visited on 2015-05-17).

[Mog91] Moggi, E. "Notions of computation and monads". In: *Information and Computation* 93.1 (1991), pp. 55–92. DOI: 10.1016/0890-5401(91)90052-4.

[Ode15] Odersky, M. *Scala – where it came from, where it's going*. Presentation. Scala Days San Francisco, 2015. URL: https://www.parleys.com/tutorial/scala-where-came-from-where-its-going (visited on 2015-04-17).

[OSV08] Odersky, M.; Spoon, L.; Venners, B. *Programming in Scala*. 2nd ed. Artima, 2008.

[Ode+12] Odersky, M. et al. *SIP-15 – value classes*. (Completed). 2012. URL: http://docs.scala-lang.org/sips/completed/value-classes.html (visited on 2015-05-17).

[Ode+14] Odersky, M. et al. *Scala language specification, version 2.11*. 2014. URL: http://www.scala-lang.org/files/archive/spec/2.11/ (visited on 2015-04-23).

[OMO10] Oliveira, B. C.; Moors, A.; Odersky, M. "Type classes as objects and implicits". In: *SIGPLAN Not.* 45.10 (2010), pp. 341–360. DOI: 10.1145/1932682.1869489.

[TaPL02] Pierce, B. C. *Types and programming languages*. Cambridge: MIT Press, 2002.

[R15] R Core Team. *R: A language and environment for statistical computing*. R Foundation for Statistical Computing. Vienna, 2015. URL: https://www.R-project.org.

[Sam72] Sammet, J. E. "Programming languages: history and future". In: *Communications of the ACM* 15.7 (1972), pp. 601–610. DOI: 10.1145/361454.361485.

[SBT15] *Sbt reference manual*. Version 0.13. 2015. URL: http://www.scala-sbt.org/0.13/docs/sbt-reference.pdf (visited on 2015-06-22).

[Sel15] Selivanov, Y. *PEP 0492 – coroutines with async and await syntax*. Python Software Foundation, 2015. URL: https://www.python.org/dev/peps/pep-0492 (visited on 2015-11-01).

[Sha] Shabalin, D. *Quasiquotes: introduction*. URL: http://docs.scala-lang.org/overviews/quasiquotes/intro.html (visited on 2015-04-23).

[SBO13] Shabalin, D.; Burmako, E.; Odersky, M. *Quasiquotes for Scala*. Tech. rep. École Polytechnique Fédérale de Lausanne, 2013. URL: http://infoscience.epfl.ch/record/185242/files/QuasiquotesForScala.pdf (visited on 2015-06-09).

[Spio1] Spinellis, D. "Notable design patterns for domain-specific languages". In: *Journal of Systems and Software* 56.1 (2001), pp. 91–99. DOI: 10.1016/S0164-1212(00)00089-3.

[Stroo] Strachey, C. "Fundamental concepts in programming languages". In: *Higher-Order and Symbolic Computation* 13.1–2 (2000), pp. 11–49. DOI: 10.1023/A:1010000313106.

[Sue] Suereth, J. *String interpolation*. URL: http://docs.scala-lang.org/overviews/core/string-interpolation.html (visited on 2015-04-23).

[Sym12] Syme, D. *The F# 3.0 language specification*. Microsoft Research. 2012. URL: http://fsharp.org/specs/language-spec/3.0/FSharpSpec-3.0-final.pdf (visited on 2015-11-26).

[Tap15] Tappler, M. "Symbolic input output conformance checking of action system models". Master's thesis. Graz University of Technology, 2015.

[Tho91] Thompson, S. *Type theory and functional programming*. Addison Wesley, 1991.

[Tor10] Torgerson, M. *Asynchronous functions in C#*. Microsoft, 2010.

[VS15] Valverde, S.; Solé, R. V. "Punctuated equilibrium in the large-scale evolution of programming languages". In: *Journal of The Royal Society Interface* 12.107 (2015). DOI: 10.1098/rsif.2015.0249.

[Voi13] Voitot, P. *Play2 JSON interpolation & pattern matching*. 2013. URL: http://mandubian.com/2013/07/04/json-interpolation-pattern-matching/ (visited on 2015-04-22).

[WB89] Wadler, P.; Blott, S. "How to make ad-hoc polymorphism less ad hoc". In: *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL)*. ACM, 1989, pp. 60–76. DOI: 10.1145/75277.75283.

[Wilo4] Wile, D. "Lessons learned from real DSL experiments". In: *Science of Computer Programming* 51.3 (2004), pp. 265–290. DOI: 10.1016/j.scico.2003.12.006.

A Example Programs

A.1 SimpleRocket in Original Syntax

```
def Rocket
{
  types
  {
    MyNat = [0..10];
    Mode = Air | Ground | Destroyed;
    Bool = [0..1];
  }

  state
  {
    engine: Bool;
    mode: Mode;
    pos_x: MyNat;
    pos_y: MyNat;
  }

  init
  {
    engine := 0;
    mode := Ground;
    pos_x := 0;
    pos_y := 0;
  }

  actions
  {
    powerOn() if engine == 0 && !(mode == Destroyed) then
    {
      engine := 1;
    };

    powerOff() if engine == 1 && mode == Ground then
    {
      engine := 0;
    };
```

```
    start() if engine == 1 && mode == Ground then
    {
      mode := Air;
    };

    land() if mode == Air then
    {
      mode := Ground;
    };

    move(x:MyNat, y:MyNat) if mode == Air && engine == 1 then
    {
      pos_x := pos_x + x;
      pos_y := pos_y + y;
    };

    destroy() if mode == Air then
    {
      mode := Destroyed;
      engine := 0;
    };
  }
}
```

A.2 SimpleRocketNoDSL

```
abstract class SimpleRocketNoDSL
  extends ActionSystem
  with BaseTypes {

  type State = Int

  val F = 0
  val T = 1
  val Air = 0
  val Ground = 1
  val Destroyed = 2

  val engine = Variable('engine)
  val mode = Variable('mode)
  val pos_x = Variable('pos_x)
  val pos_y = Variable('pos_y)

  initialize (
    Assignment(engine, Constant(F)),
    Assignment(mode, Constant(Ground)),
    Assignment(pos_x, Constant(0)),
```

```
    Assignment(pos_y, Constant(0))
  )

  addAction(Action('PowerOn,
    And(Predicate2("==", engine, Constant(F)), Predicate2("!=", mode,
      ↳Constant(Destroyed))),
    Seq(Assignment(engine, Constant(T))))
  )

  addAction(Action('PowerOff,
    And(Predicate2("==", engine, Constant(T)), Predicate2("==", mode,
      ↳Constant(Ground))),
    Seq(Assignment(engine, Constant(F))))
  )

  addAction(Action('Start,
    And(Predicate2("==", engine, Constant(T)), Predicate2("==", mode,
      ↳Constant(Ground))),
    Seq(Assignment(mode, Constant(Air))))
  )

  addAction(Action('Land,
    Predicate2("==", mode, Constant(Air)),
    Seq(Assignment(mode, Constant(Ground))))
  )

  addAction(Action('Destroy,
    Predicate2("==", mode, Constant(Air)),
    Seq(Assignment(engine, Constant(F)), Assignment(mode,
      ↳Constant(Destroyed))))
  )

  addAction(Action('Move,
    And(Predicate2("==", engine, Constant(T)), Predicate2("==", mode,
      ↳Constant(Air))),
    Seq(Assignment(pos_x, Application("+", pos_x, Variable('dx))),
      Assignment(pos_y, Application("+", pos_y, Variable('dy)))),
    Seq(Variable('dx), Variable('dy)))
  )
}
```

A.3 SimpleRocket

```
abstract class SimpleRocket
  extends ActionSystem
  with GherkinSyntax with ExternalEffects
  with BaseTypes
```

```
with As2BmcTranslator {

type State = Int

val F = 0
val T = 1
val Air = 0
val Ground = 1
val Destroyed = 2

val engine = statevar('engine)
val mode = statevar('mode)
val pos_x = statevar('pos_x)
val pos_y = statevar('pos_y)


initialize (
  engine := F,
  mode := Ground,
  pos_x := 0,
  pos_y := 0
)


when('PowerOn) given engine === F and mode =!= Destroyed then_do (
  engine := T
)

when('PowerOff) given engine === T but mode === Ground then_do (
  engine := F
)

when('Start) given engine === T && mode === Ground then_do (
  mode := Air
)

when('Land) given mode === Air then_do (
  mode := Ground
)

when('Destroy) given mode === Air then_do (
  mode := Destroyed,
  engine := F
)

when('Move('dx, 'dy)) given mode === Air && engine === T then_do (
  pos_x := pos_x + 'dx,
  pos_y := pos_y + 'dy
)
```

```
}
```

A.4 ExtendedSimpleRocket

```
abstract class ExtendedSimpleRocket
  extends ActionSystem
  with GherkinSyntax with ExternalEffects
  with BaseTypes {

  type State = Int

  val F = 0
  val T = 1
  val Air = 0
  val Ground = 1
  val Destroyed = 2

  val engine = statevar('engine)
  val mode = statevar('mode)
  val pos_x = statevar('pos_x)
  val pos_y = statevar('pos_y)


  initialize (
    engine := F,
    mode := Ground,
    pos_x := 0,
    pos_y := 0
  )


  var cnt = 0

  when('PowerOn) given engine === F and mode =!= Destroyed then_do (
    engine := T
  )

  when('PowerOff) given engine === T but mode === Ground then_do (
    engine := F
  )

  when('Start) given engine === T && mode === Ground then_do (
    mode := Air
  )

  // `given' and `when' can be swapped
  given (mode === Air) when 'Land then_do (
    mode := Ground
```

```
  )

  // external statement, closing over local variable
  when('Destroy) given mode === Air then_do (
    mode := Destroyed,
    engine := F,

    externally(println(s"E: BOOM * $cnt")),
    externally(cnt += 1)
  )

  // multiple actions with the same name
  when('Destroy) given mode === Ground then_do (
    mode := Destroyed,
    engine := F,

    externally(println("How could that happen?"))
  )

  // external statement accessing the environment of the action
  // at the time of evaluation
  when('Position('x, 'y)) given mode === Air && engine === T then_do (
    pos_x := 'x,
    pos_y := 'y,

    // thinkable alternative syntaxes:
    // - externally { implicit env => ...} (explicitly passing the
    //   environment by an implicit lambda)
    // - externally('x, 'y) { ... } (like C++ lambda capture)
    externally {
      println(s"E: x = ${'x.value}, y = ${'y.value}, pos_x =
        ↳ ${'pos_x.value}")
      if ('x.value < 0 || 'y.value < 0) {
        println("Aborting: x < 0 || y < 0.")
        abort
      }
    }
  )

  when('Move('dx, 'dy)) given mode === Air && engine === T then_do (
    pos_x := pos_x + 'dx,
    pos_y := pos_y + 'dy
  )
}
```

Colophon

This document was typeset using the pdfLaTeX typesetting system, with the memoir document class. The body text is set in 11 pt Linux Libertine, enhanced by the microtype package. Other fonts include Biolinum, Inconsolata, and GFS Neohellenic. The drawings are typeset using TikZ/PGF, and source code examples are formatted by the listings package.

The document source has been written in Emacs with AUCTeX mode, using TeXworks as PDF viewer.