Languages & translators: Domain-specific languages

Gilles Peiffer (24321600), Liliya Semerikova (64811600)

Abstract

This paper gives some more insight into the domain-specific language we designed for the third project of LINGI2132. Several key design choices are explained, and more information is given about the strengths and weaknesses of our implementation. We also briefly cover possible further improvements which one could make to the DSL in order to enable more functionality or improve performance/readability.

I. Introduction

Domain-specific languages (DSL) are an interesting branch of programming language design that has been getting a lot of attention in the literature recently. Many general-purpose languages (GPL) are not suited to the specificities of several domains, and the ability to add custom language constructs to an existing language has a large amount of pull on software engineers choosing a language for their project. One GPL allowing the use of DSLs is Scala, which is based on the JVM. This paper contains the description of a proof-of-concept DSL called[1] "**D**sl for **U**ltimate **BR**owser-based entert**A**inment functionalit**Y**", or "**DUBRAY**" for short, which allows one to build simple browser-based games using Scala.js.

In order to do that, we developed a simple implementation of the well-known Snake game, which uses our DSL. Section II gives an overview of the various functionalities of our DSL, with documented examples. In Section III, we describe various strengths and weaknesses of our DSL. Finally, Section IV gives various further improvements which could be make to extend the existing DSL with more functionality.

II. Functionalities

A. First DSL

Our DSL is heavily inspired from the DSL given in the first part of the project, which was based around various `Shape`s (`Rectangle` and `Circle`) and how one could add them together into so-called "`ComposedShape`s", containing a `List` of `Shape`s.

Several properties were defined for these `Shape`s, such as their color, the stroke width with which they should be drawn, or geometric attributes such as the width, height and radius.

B. Design Choices

Our final DSL is based around entities called `Spot`s, which represent an entity which can be drawn on a canvas. Structurally, they are similar to the `Shape`s of the previous exercise. `ComposedSpot`s and `ComposedSpot2D`s are one-dimensional and two-dimensional arrays of `Spot`s, and can be used to represent a group of related `Spot`s (such as, for example, the body parts of a snake).

C. Properties

Many of the properties which make Scala well-suited for building DSLs were used, such as traits, monads, strong and static typing, type inference, implicits, currying, closures, etc.

1) Traits

Our implementation uses traits in order to define what a `Spot` represents (in our case, it has a position, a `move` method to change its position, and a `change` method to change one its properties). Additionally, a second trait is used to represent the attributes of a `Spot`: `SpotAttributes`, i.e. its color and stroke width.

```
sealed trait Spot {
        type A <: Spot
        val position: Point = Point(0, 0)
        def change(property: CanvasElementModifier[A]): Unit
        def move(p: Point): Unit = {
                position.x = p.x
                position.y = p.y
        }
}
```

[1] Another possible name is "**SC**ala-in**H**erited **A**id for **U**ser-defined **S**oftware", or "**SCHAUS**" for short.

```
trait SpotAttributes {
        var color = "black"
        var strokeWidth = 1
        def color(col: String): Unit = color = col
        def strokeWidth(n: Int): Unit = strokeWidth = n
}
```

Traits are also used when defining entities which modify these properties, so-called **CanvasElementModifier**s, which must define a `change` method as well.

```
trait CanvasElementModifier[-ApplyOn <: Spot] {
        def change(x: ApplyOn): Unit
}
```

2) Monads

ComposedSpot defines a `map` and `flatMap` method, and is thus a monad.

```
def map(f: Spot => Spot): ComposedSpot[Spot] = ComposedSpot(l.map(f))
def flatMap(f: Spot => Iterable[Spot]): ComposedSpot[Spot] = ComposedSpot(l.flatMap(f))
```

3) Strong/static Typing and Type Inference

All throughout our code, explicit and implicit typing are used. Scala requires one to use explicit types in some situations, such as function arguments, but in other places, where possible, the type is inferred at run-time. Examples of this are shown in the listings above. Our DSL also makes use of this in order to detect type mismatches.

4) Implicits

Implicits are used in our demo DSL to combine **ComposedShape**s with different element types into one list with a common supertype (**Shape**, in this case).

```
implicit def and[U <: Shape](shape: Array[U]): ComposedShape[Shape] = {
        ComposedShape(s.toList) and ComposedShape(shape.toList)
}
```

5) Currying

In **ComposedSpot2D**, the data is stored inside an array of arrays (functionally, this is thus equivalent to a two-dimensional matrix). In order to allow nicer syntax when accessing the members of such a grid, we implemented currying, as follows (where `spots` contains the matrix):

```
case class ArrayMapper(i: Int) {
        def apply(j: Int): Spot = spots(i)(j)
        def update(j: Int, v: T): Unit = spots(i)(j) = v
        def indices: Range = spots(i).indices
}
def apply(i: Int): ArrayMapper = ArrayMapper(i)
```

With this, elements of the grid can be accessed as such: `grid(i)(j)`.

6) Closures

Writing functional code is often simplified by the use of closures, as in the following examples, which checks whether a certain position is contained in a **ComposedSpot**:

```
def contains(p: Point): Boolean = l.map(_.position).contains(p)
```

D. Examples of Use

Below, we give several examples of usage of the DSL. Note that for brevity, the code has been slightly modified (though it serves the exact same function).

1) Defining a grid

The following code extract defines a grid for the Snake game:

```
// Walls.
val wallTop = ComposedSpot(Array.tabulate(w)(i => Wall(Point(0, i), size)))
val wallRight = ComposedSpot(Array.tabulate(h - 2)(i => Wall(Point(i + 1, 0), size)))
val wallBottom = ComposedSpot(Array.tabulate(w)(i => Wall(Point(h - 1, i), size)))
val wallLeft = ComposedSpot(Array.tabulate(h - 2)(i => Wall(Point(i + 1, w - 1), size)))
```

Figure 1: Screenshots from the games we implemented.

```
// Field.
val field = ComposedSpot2D(Array.ofDim[Empty](h, w))
// Define the white squares that make up the playing field.
for (i <- 0 until h; j <- 0 until w) {
        field(i)(j) = Empty(Point(i + 1, j + 1), size)
}
field change Color(emptyColor)
// Define the grid, containing both the walls and the field.
val grid = Grid(w, h, wallTop, wallBottom, wallRight, wallLeft, field)
```

We start by defining the walls, then the playing field (for which we use the `change` method in order to color it in white), and finally the grid, which combines both. It also shows an example of currying, in how the playing field is built.

2) Defining a snake

The following code extract shows how to define a snake, and how to make it appear on the grid.

```
// The snake (starts off at size 1, in the middle of the playing field).
val middle = Point((h - h%2) / 2, (w - w%2) / 2)
val snake = ComposedSpot[Snake](Array(Snake(middle, size)))
var direction = Point(1, 0) // Direction of the snake (initially, to the right).
grid.spots(middle.x.toInt)(middle.y.toInt) = snake.head
snake change Color(snakeColor)
```

The snake should start off in the middle of the grid, moving to the right initially. We also change its color to be green using the DSL.

3)

.

E. Screenshots

Figure 1 shows some screenshots from the games we wrote using our DSL.

III. Strengths and Weaknesses

As building a good DSL is partially an art, there is no construction which is optimal from all points of view. In our DSL, simplicity and clarity were privileged, meaning performance is not always optimal. Where these two goals were conflicting, the cleanest solution was chosen, unless there was a real performance bottleneck. An example of this is the use of functional programming where applicable, since immutable states brings some guarantees one cannot expect to have when using mutable state.

According to this logic, there is no clear way to define strengths and weaknesses, as every end-user will have different requirements. Since the goal is not to write an API to build a snake game, our code is not always as short as it could be, and writing a game with it still requires some boilerplate code. This, however, is by design, as our DSL should also be usable by someone wanting to write a game such a Pong or Tetris. In fact, to demonstrate this versatility, we decided to also write Pong with our DSL.

On the other hand, due to the constraints of the project, we were not able to use external libraries, which made it impossible to deal with type erasure (which itself is a consequence of the JVM and the goal of Scala developers to maintain interoperability with Java). Due to this, several parts of the code are not as clean as one would like them to be, however this complexity is mostly hidden from the end user. A possible solution is explored in Section IV.

IV. Further Improvements

The best way to go about further improving our DSL would be to try to write more games, and see which constructs would be useful in general. As the point of a DSL is only to make life easier for programmers, rather than provide a set of very specific tools which can only be used in a limited number of situations, it is not a good idea to add all functions one writes for a given game to the DSL.

One other point which could be improved is avoiding type erasure, which can be done relatively easily using `TypeTag`s. However, as mentioned in Section III, this requires additional imports which are not allowed in the context of this project. Another solution would be to use wrapper classes, but this quickly becomes very ugly when it has to be used more often.

Finally, there are definitely ways to further improve the coding style, as both authors are relatively new to Scala. Doing so might improve performance and readability in some scenarios.

V. Conclusion

Writing a domain-specific language is not a simple task, and there is no solution which is optimal in all aspects. When writing simple, browser-based games, performance is often not an issue, and writing simple code is more important. For this purpose, we designed the DUBRAY/SCHAUS DSL in Scala, which makes this task easier.

Two games were implemented using our DSL, though many other options are possible: Snake, and Pong. This shows that our DSL does make the task of writing a simple game in Scala easier.