FH Vorarlberg
University of Applied Sciences

Translating the concepts of Haskell into the object-oriented programming language Java

Written Elaboration

Vorarlberg University of Applied Sciences
Computer Science MSc

Supervised by
Jonathan Thaler, Ph.D

Submitted by
Dominik Böckle,
Dominic Luidold

Dornbirn, January 2022

Contents

List of Abbreviations **IV**

1 Introduction **1**

2 Implementation of the various concepts **2**

2.1 Immutable Data 2
2.2 Type Variables 3
2.3 Higher-Order Functions 4
2.4 Lambda Expressions 5
2.5 Currying 6
2.6 Function Composition and Streaming 7
2.6.1 Function Composition 7
2.6.2 Streaming 8
2.7 Algebraic Data Types 9
2.8 Pure and Impure Side Effects 11

Bibliography **13**

List of Abbreviations

ADT Algebraic Data Types

OOP Object-oriented programming

1 Introduction

This written elaboration contains various concepts of functional programming that are covered in the lecture "Concepts of Higher Programming Languages" in the first semester Master of Science at the Vorarlberg University of Applied Sciences. The concepts and patterns covered in this course are implemented in this paper using the Object-oriented programming (OOP) language Java. Each implementation is complemented by a short explanation.

In addition, a `StateMonad` is implemented in Java and an example of how to use the Monad implemnetation is provided.

2 Implementation of the various concepts

This chapter covers the implementation of eight functional programming concepts and their respective OOP counterparts written in Java. The concepts and patterns are each accompanied by a brief explanation as well as a working Java example.

2.1 Immutable Data

Immutable Data is a principle that states that once data has been created, it subsequently cannot be changed. In general, data and objects should only be mutable if there is a valid reason for doing so.

```
1  public class ImmutableIntegerPair {
2      private final int x;
3      private final int y;
4
5      ImmutableIntegerPair(int x, int y) {
6          this.x = x;
7          this.y = y;
8      }
9
10     // Getters for variable 'x' and 'y'
11 }
```

Sourcecode 1: Example of an immutable data structure

Sourcecode 1 on page 2 shows how an immutable data structure may look. The variables `x` and `y`, which contain the data, can no longer be modified after the value has initially been set as a result of the `final` keyword. Additionally,

the variables cannot be changed from outside the class as there are no setters present.

There is, however, still a possibility to make the data structure mutable: Extending the `ImmutableIntegerPair` allows one to extend the functionality of the data structure and store mutable data. To prevent this, the keyword `final` can be used again, this time at class level as opposed to variable level.

Furthermore, there are also ways to make lists immutable when the list data type is the preferred/required way of storing data. Java itself provides a static method `Collections::unmodifiableList()` which prevents the data from being modified. However, the data type that is stored in the list must also be immutable to achieve immutability.

2.2 Type Variables

As far as the concept of *Type Variables* is concerned, Java - being a strongly typed programming language - enables the typing of variables and the enforcement of a certain type throughout the programme flow in various ways:

- Each variable has to have an accompanying data type when getting declared, e.g. `int x = 1;`
- Java natively supports Generics (as can be seen in *Sourcecode 7* on page 10) which allows types to be enforced based on the specific use case at compile time. Constructs such as `<? extends Integer>` additionally allow the restriction of which types are "allowed".
- Checking types at run time is possible using `instanceof` when there are several types possible at one point in a time, for example when having a common interface and different classes inheriting that interface.

2.3 Higher-Order Functions

A *Higher-Order Function* is a function that either takes a function as an argument or has a function as a return value. Implementing this concept has been possible since Java 8 - an example for that is shown in *Sourcode 2* on page 4. The method `camelize` capitalises the first character of a string and returns the required method/function calls when called. This method also makes use of *Lambdas*, which is explained in more detail in a subsequent chapter.

```
1  import java.util.Arrays;
2  import java.util.function.Function;
3
4  /**
5   * @see <a href="https://medium.com/@knoldus/functional-java-lets-understan
   ↪  d-the-higher-order-function-1a4d4e4f02af">Source: Knoldus Inc. on
   ↪  Medium</a>
6   * @see <a href="https://stackoverflow.com/a/15200056/5232876">Source:
   ↪  n1ckolas on Stackoverflow</a>
7   */
8  public class HigherOrderFunctions {
9      public static void main(String[] args) {
10         Function<Integer, Long> addOne = add(1L);
11
12         System.out.println(addOne.apply(1)); // prints 2
13
14         Arrays.asList("test", "new")
15                 .parallelStream()  // suggestion for execution strategy
16                 .map(camelize)     // call for static reference
17                 .forEach(System.out::println);
18     }
19
20     private static Function<Integer, Long> add(long l) {
21         return (Integer i) -> l + i;
22     }
23
24     private static final Function<String, String> camelize = (str) ->
       ↪  str.substring(0, 1).toUpperCase() + str.substring(1);
25 }
```

Sourcecode 2: Example of a Higher-Order Function in Java 8

2.4 Lambda Expressions

The method `listLambdas` in *Sourcode 3* on page 3 increments the `Integer` values stored in a `List` by one. This is achieved by the rather short code fragment `map(x -> ++x)`.

The same functionality can be achieved be creating an anonymous `Function` instance within the `map` method (not shown in this example), which has been the recommended and only way until to Java 8 to implement such functionality. However, Lambda Expressions have the advantage of code that is usually more readable and requires less boilerplate code.

```
1  import java.util.List;
2  import java.util.stream.Collectors;
3
4  public class LambdaExpressions {
5      public static List<Integer> listLambdas(List<Integer> list) {
6          return list.stream()
7                  .map(x -> ++x)
8                  .collect(Collectors.toList());
9      }
10 }
```

Sourcecode 3: Example of a Lambda Expression

2.5 Currying

Currying is the conversion of a function with multiple arguments into a sequence of functions with one argument each. *Sourcecode 4* on page 6 is inspired by [Rob19] in which multiplication was performed by currying. This is emulated with the function `multCurry` where the multiplications can be concatenated.

```
1  import java.util.function.Function;
2
3  /**
4   * @see <a href="https://www.geeksforgeeks.org/currying-functions-in-java-w
   ↪  ith-examples/">Source:
   ↪  GeeksForGeeks</a>
5   * @see <a href="http://baddotrobot.com/blog/2013/07/21/curried-functions/"
   ↪  >Source: Toby Weston on
   ↪  bad.robot</a>
6   */
7  public class Currying {
8      public static int multNormal(int a, int b) {
9          return a * b;
10     }
11
12     public static Function<Integer, Function<Integer, Integer>> multCurry()
       ↪  {
13         return x -> y -> x * y;
14     }
15
16     public static void main(String[] args) {
17         System.out.println(multNormal(1, 5)); // prints 5
18         System.out.println(multCurry().apply(1).apply(5)); // prints 5
19     }
20 }
```

Sourcecode 4: Example of Currying

2.6 Function Composition and Streaming

2.6.1 Function Composition

Function Composition is applying the pattern of combining multiple separate functions into a single function. The output of the first function is then used as input for the second function and so on. To be able to demonstrate this in an example, *Sourcecode 5* on page 7 containts the functions `doubleInt` and `substractOne` - both based on the use of Lambda Expressions.

An important part of Function Composition is that order matters. While using `compose(<fn>)` results in the supplied function (`substractOne`) being executed before the initial function (`doubleInt`), using `andThen(<fn>)` has the opposite result.

```
1  import java.util.function.Function;
2  import java.util.function.Predicate;
3
4  /**
5   * @see <a href="https://functionalprogramming.medium.com/function-composit
   ↪  ion-in-java-beaf39426f52">Source: Dimitris Papadimitriou on
   ↪  Medium</a>
6   */
7  public class FunctionComposition {
8      public static void main(String[] args) {
9          Function<Integer, Integer> doubleInt = t -> t * 2;
10         Function<Integer, Integer> subtractOne = t -> t - 1;
11
12         var firstSubtractOneThenDouble = doubleInt.compose(subtractOne);
13         var firstDoubleThenSubtractOne = doubleInt.andThen(subtractOne);
14
15         System.out.println(firstDoubleThenSubtractOne.apply(2)); // prints 3
16         System.out.println(firstSubtractOneThenDouble.apply(2)); // prints 2
17     }
18 }
```

Sourcecode 5: Example of Function Composition and that order matters

2.6.2 Streaming

A *Stream* or *Streaming* represents a sequence of objects that are accessed in sequential order. One of the best known and most frequently used Streams in Java is probably the `FileStream`, which is used to read files and file contents. *Sourcecode 6* on page 8 shows this in a small, but only theoretical example.

```
1  import java.io.IOException;
2  import java.nio.charset.StandardCharsets;
3  import java.nio.file.Files;
4  import java.nio.file.Path;
5  import java.nio.file.Paths;
6  import java.util.stream.Stream;
7
8  public class Streaming {
9      // Minimal Stream
10     private final Stream<String> streamEmpty = Stream.empty();
11
12     public static void main(String[] args) {
13         try {
14             Path path = Paths.get("C:\\file.txt");
15             Stream<String> streamOfStrings = Files.lines(path);
16             Stream<String> streamOfStringsWithCharset = Files.lines(path,
               ↪  StandardCharsets.UTF_8);
17         } catch (IOException e) {
18             e.printStackTrace();
19         }
20     }
21 }
```

Sourcecode 6: Example of Streaming.

2.7 Algebraic Data Types

An Algebraic Data Types (ADT) consists of several variants or flavors, similar to a Java enum, but the different flavors can have different properties or methods.

To summarize again what we expect from an ADT: The expressions have different numbers and types of properties and methods. At compile time it is checked that all expressions are also considered.

Unfortunately, Java does not support ADT from scratch, so we have to implement this ourselves. For this, as described in [MAI], the design patterns Visitor and Sealed Class are helpful. In Java 15 the sealed class should be automatically integrated but in Sorce code it was still implemented manually without the sealed keyword.

Sourcecode 7 shows a Java implementation of the Alive or Date State from the lecture notes [1].

[1] lecture notes: https://homepages.fhv.at/thjo/lecturenotes/concepts/declaring-types.html#algebraic-data-types-1 visited on 2022/01/06

```java
public abstract class ADT {


    private ADT() {
    }

    public static final class Alive extends ADT {
        private Alive() {
        }

        public static final Alive INSTANCE = new Alive();

        public <T> T isAlive(IsAliveChecker<T> visitor) {
            return visitor.isAlive(this);
        }
    }

    public static final class Dead extends ADT {
        public Dead() {
        }

        public <T> T isAlive(IsAliveChecker<T> visitor) {
            return visitor.isAlive(this);
        }
    }

    // The IsAliveChecker Interface ensures that all possible expressions
    // are taken into account, provided that the interface
    // is also used in the implementation of the business logic.

    // The Sealed Classs pattern uses an abstract class
    // with a private constructor so that concrete classes
    // can only be created as inner classes within the ADT.
    public interface IsAliveChecker<T> {
        T isAlive(Alive m);

        T isAlive(Dead l);
    }

    public abstract <T> T isAlive(IsAliveChecker<T> visitor);

    public static void main(String[] args) {
        var state = new Dead();
        var bool = state.isAlive(new IsAliveChecker<Boolean>() {

            public Boolean isAlive(ADT.Alive a) {
                return true;
            }
            public Boolean isAlive(ADT.Dead a) {
                return false;
            }
        });
        // By using IsAliveChecker here and not if-instanceOf-else,
10      // we can be sure at compile time that all possible cases have been considered.
        System.out.println(bool);

    }

}
```

Sourcecode 7: Dead or Alive Example to demonstrate ADT.

2.8 Pure and Impure Side Effects

Pure: Side effects are effects which are influenced by the Pure calculation and are always deterministic because they do not communicate with the outside world. *Sourcecode 8* shows the change of the "State" in the method Pure.
Impure: are all effects which interact with the outside world this is declared in *Sourcecode 8* with the method Impure which waits for a user action

```java
import java.io.IOException;
import java.nio.charset.Charset;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.Scanner;
import java.util.stream.Stream;

public class SideEffects {
    public static int State=0;

    public static void main(String[] args) {
        Impure();

        System.out.println(SideEffects.State);
        var s = new SideEffects();
        s.Pure(10);
        System.out.println(SideEffects.State);

    }

    private static void Impure(){
        System.out.println("Enter string ");

        // Using Scanner for Getting Input from User
        Scanner in = new Scanner(System.in);

        String s = in.nextLine();
        System.out.println("You entered string " + s);

    }

    private void Pure(int x){
        State = x;

    }
}
```

Sourcecode 8: Example for Side Effects.

Bibliography

[MAI] NICOLAI MAINIERO. *Algebraic Data Types in Java - sidion*. URL: `https://www.sidion.de/lernen/sidion-labor/blog/algebraic-data-types-in-java.html` (visited on 01/06/2022).

[Rob19] Dan Robertson. *currying - How to curry functions in Haskell*. 2019. URL: `https://stackoverflow.com/questions/54006099/how-to-curry-functions-in-haskell` (visited on 01/06/2022).