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

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

For the area of type variables there is the possibility of generic programming in Java. Here one can restrict like with Haskell which types are permitted. This restriction can be made in Java by interfaces or classes and write for example <? extends Integer>.

2.3 Higher-Order Functions

A higher order function is a function that takes a function as an argument or returns a function. This is possible since Java 8. *Sourcecode 2* shows an example code where the method camelize turns the first characters of a string into a capital letter. In this method you can also see the use of lambdas which will be explained in more detail in one of the following sections

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.function.Function;
import java.util.stream.Collectors;

// Source:
//
↪  https://medium.com/@knoldus/functional-java-lets-understand-the-higher-order-function-1a4d
//
↪  https://stackoverflow.com/questions/15198979/lambda-expressions-and-higher-order-functions
public class HigherOrderFunctions {
    public static void main(String[] args) {
        Function<Integer, Long> addOne = add(1L);

        System.out.println(addOne.apply(1)); //prints 2

        Arrays.asList("test", "new")
                .parallelStream()  // suggestion for execution strategy
                .map(camelize)     // call for static reference
                .forEach(System.out::println);
    }

    private static Function<Integer, Long> add(long l) {
        return (Integer i) -> l + i;
    }

    private static Function<String, String> camelize = (str) ->
            str.substring(0, 1).toUpperCase() + str.substring(1);

}
```

Sourcecode 2: Example for Higher Order Function

2.4 Lambda Expressions

```
1  import java.util.List;
2  import java.util.stream.Collectors;
3
4  public class Lambda {
5      public static List<Integer> listLambdas(List<Integer> list) {
6          // Lambdas
7          return list.stream()
8                  .map(x-> ++x)
9                  .collect(Collectors.toList());
10     }
11 }
```

Sourcecode 3: Example for Lambda Expressions.

With the method listLambdas in *Sourcecode 3* you can see how with a lambda function the values of the list are increased by 1. In the past, anonymous classes often had to be used instead of lambdas in Java, but this has been curbed somewhat with Java 8 and the lambdas and usually makes the code more readable.

2.5 Currying

Currying is the conversion of a function with multiple arguments into a sequence of functions with one argument each. *Sourcecode 4* is inspired by [Rob19] in which multiplication was done by currying. This is emulated with the function multcurry where the multiplications can be strung together.

```
1  import java.util.Arrays;
2  import java.util.function.Function;
3  // Example 2 https://www.geeksforgeeks.org/currying-functions-in-java-with-examples/
4  //Source: http://baddotrobot.com/blog/2013/07/21/curried-functions/
5  public class Currying {
6      public static int multNormal(int a, int b) {
7          return a * b;
8      }
9
10     public static Function<Integer, Function<Integer, Integer>> multcurry() {
11         return x -> y -> x * y;
12     }
13
14     public static void main(String[] args) {
15         var mult1 = multNormal(1,5); //= 5
16
17         System.out.println(mult1); //prints 5
18         System.out.println(multcurry().apply(1).apply(5)); // prints 5
19
20     }
21 }
```

Sourcecode 4: Example for multiplication with Currying.

2.6 Function Composition and Streaming

2.6.1 Function Composition

Function composition is the combination of two functions into a new function. You simply take the output of the first function and use it as input for the second function. To be able to demonstrate this in *Sourcecode 5* the functions doubleing and remove1 were expressed by means of Lambda and used for an example. Here it is remarkable that with .compose(<function>) the function is put in front and with andThen(<function>) the function is put after.

```
1  import java.util.function.Function;
2
3  // Source:
   ↪  https://functionalprogramming.medium.com/function-composition-in-java-beaf39426f52
4  // Functional composition refers to a technique where multiple functions are
   ↪  combined into a single function.
5  // We can combine lambda expressions together. Java provides built-in
   ↪  support through the Predicate and Function classes.
6  // The following example shows how to combine two functions using the
   ↪  Predicate approach.
7  public class FunctionComposition {
8      public static void main(String[] args) {
9      Function<Integer, Integer> doubleing = t -> t * 2;
10     Function<Integer, Integer> remove1 = t -> t  - 1;
11     var FirstRemove1ThenDouble = doubleing.compose(remove1);
12     var FirstDoubleThenRemove1 = doubleing.andThen(remove1);
13     System.out.println(FirstDoubleThenRemove1.apply(2)); // result 3
14     System.out.println(FirstRemove1ThenDouble.apply(2)); // result 3
15     }
16 }
```

Sourcecode 5: Example for Function Composition and that order can be Important.

2.6.2 Streaming

A stream represents a sequence of objects that can be accessed in sequential order. One of the better known streams in Java is probably the filestream this is used in *Sourcecode 6*.

```
1  import java.io.IOException;
2  import java.nio.charset.Charset;
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     Stream<String> streamEmpty = Stream.empty();
11     public static void main(String[] args) {
12         try {
13             Path path = Paths.get("C:\\file.txt");
14             Stream<String> streamOfStrings = Files.lines(path);
15             Stream<String> streamWithCharset =
16                     Files.lines(path, Charset.forName("UTF-8"));
17         }catch(IOException e)
18         {
19             e.printStackTrace();
20         }
21
22     }
23 }
```

Sourcecode 6: Example for 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).