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

List of Source Codes

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 }
```

Source Code 1: Example of an immutable data structure

Source Code 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 *Source Code 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 }
```

Source Code 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 }
```

Source Code 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. *Source Code 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 }
```

Source Code 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, *Source Code 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 }
```

Source Code 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. *Source Code 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 }
```

Source Code 6: Example of Streaming.

2.7 Algebraic Data Types

Algebraic Data Types (ADT) consist of several variants or "flavors", similar to a Java Enum, but these different "flavors" can each have different properties and/or methods. To quickly summarise what we expect from an ADT:

- The expressions have different amounts and types of properties and methods.
- At compile time, it is checked whether all have been taken into account.

Unfortunately, Java does not support ADT natively, so we had to implement this ourselves. For this, as described in [Mai20], the *Visitor* and *Sealed Class* design patterns are helpful. Since Java 15, the `sealed` keyword is available, but the following example nevertheless includes a manually implemented *sealed* functionality. *Source Code 7* on page 10 and before shows a corresponding Java implementation of the *Dead or Alive* State from the lecture notes[1]

```
1  public abstract class AlgebraicDataTypes {
2      private AlgebraicDataTypes() {
3      }
4
5      public static final class Alive extends AlgebraicDataTypes {
6          private Alive() {
7          }
8
9          public <T> T isAlive(IsAliveChecker<T> visitor) {
10             return visitor.isAlive(this);
11         }
12     }
13
14     public static final class Dead extends AlgebraicDataTypes {
15         public Dead() {
16         }
17
18         public <T> T isAlive(IsAliveChecker<T> visitor) {
19             return visitor.isAlive(this);
```

[1]Lecture Notes, Jonathan Thaler, Ph.D (visited on 2022/01/06)

```
20          }
21      }
22
23      // The IsAliveChecker interface ensures that all possible expressions
24      // are taken into account, provided that the interface
25      // is also used in the implementation of the business logic.
26
27      // The Sealed Class pattern uses an abstract class
28      // with a private constructor so that concrete classes
29      // can only be created as inner classes within the ADT.
30      public interface IsAliveChecker<T> {
31          T isAlive(Alive m);
32
33          T isAlive(Dead l);
34      }
35
36      public abstract <T> T isAlive(IsAliveChecker<T> visitor);
37
38      public static void main(String[] args) {
39          Dead state = new Dead();
40          Boolean bool = state.isAlive(new IsAliveChecker<>() {
41
42              public Boolean isAlive(Alive a) {
43                  return true;
44              }
45
46              public Boolean isAlive(Dead a) {
47                  return false;
48              }
49          });
50
51          // By using IsAliveChecker here and not if-instanceOf-else,
52          // we can be sure at compile time that all possible cases have been
            ↪  considered.
53          System.out.println(bool);
54      }
55  }
```

Source Code 7: “Dead or Alive” example demonstrating ADT

2.8 Pure and Impure Side Effects

Pure Side Effects are effects which are solely caused by a method or calculation. They are always deterministic, as they do not communicate with the "outside world". *Source Code 8* on page 11 shows the change of the `State` within the method `pure`.

Impure Side Effects are all effects caused by interacting with the "outside world". This can be seen in *Source Code 8* within the `impure` method which awaits user action/input.

```
1  import java.util.Scanner;
2
3  public class SideEffects {
4      public static int State = 0;
5
6      public static void main(String[] args) {
7          impure();
8
9          System.out.println(SideEffects.State);
10         SideEffects s = new SideEffects();
11         s.pure(10);
12         System.out.println(SideEffects.State);
13     }
14
15     private static void impure() {
16         System.out.println("Enter string");
17
18         Scanner in = new Scanner(System.in);
19
20         String s = in.nextLine();
21         System.out.println("You entered string " + s);
22     }
23
24     private void pure(int x) {
25         State = x;
26     }
27 }
```

Source Code 8: Example of Side Effects in Java

2.9 State Monad in Java

The following section contains an implementation of the *State Monad* in Java. During the implementation of the Monad, the work of [Fau15] was of great support. A question that might arise: "What is a Monad?". Matt Fowler describes it as follows:

> Think of monads as an object that wraps a value and allows us to apply a set of transformations on that value and get it back out with all the transformations applied. [Fow15]

2.9.1 Implementing the State Monad

To develop a working State Monad in Java, both *lift* and *bind* had to be implemented as well as a `StateTuple` that stores the state as well as the values as content. *Source Code 9* on page 13 shows the class `StateMonad<S, C>` with an implementation of `liftM` which corresponds to *bind* - see Haskell: `f m = m >>= \x -> return (f x)`.

```java
package statemonad;

import java.util.function.Function;

/**
 * <a>https://faustinelli.wordpress.com/2014/04/25/the-state-monad-in-java-8-eventually/</a> Source: Marco Faustinelli
 */
public class StateMonad<S, C> {
    public final Function<S, StateTuple<S, C>> runState;

    public StateMonad(Function<S, StateTuple<S, C>> runState) {
        this.runState = runState;
    }

    public StateTuple<S, C> StatetoTuple(S state) {
        return this.runState.apply(state);
    }

    /**
     * UNIT
     */
    public static <S, C> StateMonad<S, C> UNIT(C a) {
        return new StateMonad<S, C>((S s) -> new StateTuple<S, C>(s, a));
    }

    /**
     * Promote a function to a Monad.
     * <p>
     * In Haskell:
     * liftM f m = m >>= \x -> return (f x)
     */
    public <B> StateMonad<S, B> liftM(final Function<C, StateMonad<S, B>> func) {
        return new StateMonad<S, B>((S s) -> {
            StateTuple<S, C> content = this.StatetoTuple(s);
            StateMonad<S, B> out = func.apply(content.getContent());

            return out.StatetoTuple(content.getState());
        });
    }
}
```

Source Code 9: State Monad implementation

Source Code 10 on page 14 shows the implementation representing the state and the content of the data within the State Monad.

```java
package statemonad;

public class StateTuple<S, C> {
    private S state;
    private C content;

    public StateTuple(S state, C content) {
        this.state = state;
        this.content = content;
    }

    public C getContent() {
        return content;
    }

    public void setContent(C content) {
        this.content = content;
    }

    public S getState() {
        return state;
    }

    public void setState(S state) {
        this.state = state;
    }

    @Override
    public String toString() {
        return "Lable:" + state + " Data:" + content;
    }
}
```

Source Code 10: `StateTuple` implementation

2.9.2 Implementing Tree Labelling

Source Code 11 on page 16 and before shows the implementation of a *Binary Tree* labelling, while *Source Code 12* on page 16 shows the abstract class `Tree` that defines the basic behaviour of the Binary Tree.

```
1  package statemonad;
2
3  public class Labelling {
4
5      public static void main(String[] args) {
6          // Solution from Exercise 12:
7          // tree = Node (Node (Leaf 'c') 'b' (Node (Leaf 'e') 'd' (Leaf
           ↪  'f'))) 'a' (Leaf 'g')
8          Node<Character> tree = new Node<>(new Node<>(new Leaf<>('c'), 'b',
           ↪  new Node<>(new Leaf<>('e'), 'd', new Leaf<>('f'))), 'a', new
           ↪  Leaf<>('g'));
9          tree.show();
10
11         StateMonad<Integer, Tree<StateTuple<Integer, Character>>> treeMonad
           ↪  = labelTree(tree);
12         Tree<StateTuple<Integer, Character>> result =
           ↪  treeMonad.StatetoTuple(0).getContent();
13
14         System.out.println("Labled");
15         result.show();
16     }
17
18     // labelTree :: Tree a -> Tree (a, Int)
19     private static StateMonad<Integer, Tree<StateTuple<Integer,
       ↪  Character>>> labelTree(Tree<Character> t) {
20         StateMonad<Integer, Integer> updateState = new StateMonad<>(n ->
           ↪  new StateTuple<>(n + 1, n));
21
22         // Pattern Matching would be great here..
23         if (t.isLeaf()) {
24             Leaf<Character> leaf = (Leaf<Character>) t;
25
```

```
26              return updateState.liftM((Integer state) -> new
                ↪  StateMonad<>((Integer s) -> new StateTuple<>(s, new
                ↪  Leaf<>(new StateTuple<>(state, leaf.getData()))));
27          } else {
28              Node<Character> node = (Node<Character>) t;
29              Tree<Character> oldLeft = node.getLeftChild();
30              Tree<Character> oldRight = node.getRightChild();
31
32              return labelTree(oldLeft)
33                      .liftM((Tree<StateTuple<Integer, Character>>
                        ↪  leftLabeledSubtree) -> labelTree(oldRight)
34                              .liftM((Tree<StateTuple<Integer, Character>>
                                ↪  rightLabeledSubtree) -> new StateMonad<>(
35                                      (Integer state) -> new
                                        ↪  StateTuple<>(state, new
                                        ↪  Node<>(leftLabeledSubtree, new
                                        ↪  StateTuple<>(state,
                                        ↪  node.getData()),
                                        ↪  rightLabeledSubtree)))));
36          }
37      }
38  }
```

Source Code 11: Tree labelling implementation

```
1  package statemonad;
2
3  // data Tree a = Leaf a | Node (Tree a) a (Tree a) deriving Show
4  public abstract class Tree<T> {
5      public abstract void show();
6
7      public abstract boolean isLeaf();
8  }
```

Source Code 12: `Tree` implementation

Bibliography

[Fau15] Faustinelli, Marco. *The state monad in Java 8, eventually...* Apr. 25, 2015. URL: https://faustinelli.wordpress.com/2014/04/25/the-state-monad-in-java-8-eventually/ (visited on 01/13/2022).

[Fow15] Matt Fowler. *Understanding the Optional Monad in Java 8*. Oct. 16, 2015. URL: https://medium.com/coding-with-clarity/understanding-the-optional-monad-in-java-8-e3000d85ffd2 (visited on 01/13/2022).

[Mai20] Nicolai Mainiero. *Algebraic Data Types in Java*. Nov. 3, 2020. 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*. Jan. 2, 2019. URL: https://stackoverflow.com/a/54006458/5232876 (visited on 01/06/2022).